Reducing distance computations for distance-based outliers

# Reducing distance computations for distance-based outliers

Fabrizio Angiulli[a], Stefano Basta[b], Stefano Lodi[c], Claudio Sartori[c]

[a] *DIMES Department, University of Calabria, Via P. Bucci 41C, 87036 Rende (CS), Italy. E-mail: f.angiulli@dimes.unical.it.*
[b] *Institute of High Performance Computing and Networking, Italian National Research Council, Via P. Bucci 8-9 C, 87036 Rende (CS), Italy. E-mail: stefano.basta@icar.cnr.it.*
[c] *Department of Computer Science and Engineering, University of Bologna, Viale Risorgimento 2, 40136 Bologna, Italy. E-mail: {stefano.lodi,claudio.sartori}@unibo.it.*

**Abstract**

The mining task of outlier detection is essential in many expert and intelligent systems exploited in a wide range of applications, from intrusion detection to molecular biology. In some of such applications the ability to process large amounts of data in a very short time can be critical, for instance in intrusion and fraud detection. This paper explores a solution for the optimisation of an exact, unsupervised outlier detection method by avoiding unnecessary computations, and therefore reducing the running time and making the method usable also in settings where response times are crucial. In particular, we enhance the *SolvingSet*-based approach by using a mechanism that exploits the knowledge learned during the algorithm execution and avoids a large amount of distance computations. We demonstrate the strength of the proposed solution, named *FastSolvingSet*, through both theoretical and experimental analysis.

**Keywords**

Distance-based outliers; outlier detection; parallel and distributed algorithms.

## 1. Introduction

It is well known that dealing with Big Data poses some challenges, summarised by the so-called "four v's". The problems are even harder when we

require to a system an *intelligent behaviour*, as it is for most of the so–called *data mining tasks*. In this application area, the aim is to build systems whose intelligent decisions are driven by the data, often with little or no information from the *human experts*, therefore requiring that the expertise is derived by the system directly from the data.

In this paper, we deal with the problem of *unsupervised outlier detection* and improve an already available solution considering the problems deriving from *volume* and *velocity*. In particular, we seek performance improvements exploiting the distinctive features of a method for labelling outliers to save a number of useless distance computations without introducing any approximation.

Outlier detection is a fundamental intelligent task which consists in the identification of observations which deviate substantially from the rest of the observations, raising the hypothesis that they were not generated by the same mechanism (Han et al., 2011). The usefulness of outlier detection is apparent in many application domains, in particular in those which require reaction within a predefined time, such as medical anomaly detection, sensor networks, industrial damage detection, cyber-intrusion detection, fraud detection, image processing, and textual anomaly detection (Chandola et al., 2009).

Methods for outlier detection can be *supervised*, *semi-supervised*, and *unsupervised*. In supervised methods, a set of examples labeled as normal and abnormal is available; the problem is then approached as a classification problem. In semi-supervised methods, only a small subset of the data is labeled, normal or abnormal, possibly consisting of normal or abnormal objects. Finally, unsupervised approaches to outlier detection are able to discriminate each datum as normal or abnormal when no training examples are available.

Among the unsupervised approaches, in *distance-based* methods the evaluation of the outlier status of an object involves the computation of the distances to its nearest neighbours (Knorr & Ng, 1998; Ramaswamy et al., 2000; Bay & Schwabacher, 2003; Angiulli & Pizzuti, 2005; Angiulli et al., 2006; Tao et al., 2006; Ghoting et al., 2008; Angiulli & Fassetti, 2009). The approaches differ in the way the evaluation is carried out; however, most methods associate an

object with a *weight* or *score*, which is a nondecreasing function of its $k$ nearest neighbours distances, quantifying the overall dissimilarity between the object and its neighbours.

In this work, we introduce *FastSolvingSet*, an unsupervised algorithm that represents a notable optimization of the *SolvingSet* algorithm to detect distance-based outliers (Angiulli et al., 2006).

The reason to pursue this optimisation derives from the observation than whenever an intelligent system becomes the engine of a mission critical activity the contrast between time constraints and computation times can set a threshold between useful and useless solutions. Applications of outlier detection such as intrusion and fraud detection can have very strict time constraints and a solution able to reduce the computation times without introducing approximations can be of high value.

Let us come now to some detail of our solution.

In *SolvingSet* a top-$n$ distance-based outlier in a data set is an object having weight not smaller than the $n$-th largest weight, where the weight of a data set object is computed as the sum of the distances from the object to its $k$ nearest neighbours. *SolvingSet* uses, as a base for the optimisation of the solution, the generation of an *outlier detection solving set* (in short *solving set*). The solving set is a subset $S$ of the dataset $D$ that includes a sufficient number of objects from $D$ to allow considering only the distances among the pairs in $S \times D$ to obtain the top-$n$ outliers. The solving set is a *learned model* that can be seen as a *compressed representation* of $D$. It has been shown that it can be used to predict if a novel object $q$ is an outlier or not by comparing $q$ only with the objects in $S$, instead of considering all the objects in $D$. Since the solving set contains at least the top-$n$ outliers, computing the solving set permits to solve the outlier detection task simultaneously.

As a matter of fact, we demonstrate that it is possible to find out a solving set using a number of distances among dataset objects that is remarkably lower than that one computed by the original algorithm. This outcome guarantees better performance because for the discovery of the distance-based outliers, the

cost of calculating distances generally dominates the algorithm time complexity.

The contributions of the work can be summarised as follows:

- we present a mechanism that allows to considerably reduce the computation of the distances among the pairs in $S \times D$, by exploiting only the partial results achieved. As detailed later, this mechanism bases on an efficient evaluation of a reverse triangle inequality, and it does not compromise the quality of final output, that is no approximation must be introduced;
- the proposed optimization is suitable to be used both for the sequential and the distributed version of the base algorithm (Angiulli et al., 2013);
- by using this novel strategy, we can run *SolvingSet*-based algorithms obtaining significant computation savings;
- as the secondary effect, thanks to this variant, we can efficiently run the *SolvingSet*-based approach even when the dataset (or some portion of a distributed dataset) could be so large that it would be impractical storing it on the main memory of the machine in use. Indeed in these cases the cost of calculating the distance becomes more expensive, as it requests to retrieve data from mass storage, and this strengthens the advantage of the *FastSolvingSet* algorithm with respect to the base version.

The rest of the paper is organized as follows. Section 2 overviews the literature on distance-based outlier detection. Section 3 presents preliminary definitions, recalls the SolvingSet-based approach for the outlier detection, details the *FastSolvingSet* algorithm, and provides the cost analysis. Section 4 discusses the experiments. Finally, Section 5 outlines the conclusions of the work and the future research directions.

## 2. Related work

Distance-based outlier detection methods are mostly based on the assumption of correlation between the sparseness of an object's neighbourhood and its

outlier status. The methods however differ as to the definition of neighbourhood, its relation to outlier status, generally defined in terms of an outlier score, and the approaches to possibly avoid a full scan of the dataset to compute it.

Knorr & Ng (1998) consider a neighbourhood of parametric radius $D$; an object is an outlier when at least a fraction $p$ of all the objects is contained in the complement of the neighbourhood. They present a block nested-loop (NL) algorithm with a worst-case $O(N^2)$ time complexity. Full scan of the dataset for an object is avoided by continuing with the next object when the number of objects within distance $D$ exceeds the threshold. They also present two variants of NL with $O(N)$ time complexity; the dependence on dimensionality is however exponential. With respect to *FastSolvingSet*, NL is simpler to implement, minimizes I/O block reads, and reports valid outliers according to the definition before the execution has terminated. However, distances between objects are computed multiple times, thereby increasing computation time significantly with dimensionality. In addition, the definition of outliers does not involve any ranking, and therefore enables pruning with respect to the parameters only.

Ramaswamy et al. (2000) consider a $k$-nearest neighbourhood, whereas the objects having the top $n$ distance values to their $k$-th neighbour have outlier status. The first proposed algorithm is a block nested loop join similar to NL. Its refinement is an index-based join: Full scans are avoided by recurring to spatial access methods which support minimal bounding rectangles, such as the R$^*$-tree, for the retrieval of an object's neighbourhood. Subtrees enclosed by rectangles located at a minimal distance larger than the $k$-th neighbour distance of the object are pruned. Further pruning is obtained by discarding objects having a current $k$-th distance which is exceeded by the smallest one in the top-$n$ list. Finally, the authors propose a third algorithm that partitions the data using clustering and uses minimal bounding rectangles to prune partitions which cannot contain outliers. The remaining partitions are input to an index-based join. The pruning method which excludes objects with $k$-th distance smallest than the worst known distance is similar to the one adopted in *FastSolvingSet*. Index-based or partition based pruning are not used in *FastSolvingSet*, whereas

metric space properties are not used directly in Ramaswamy et al. (2000), although they could be used in the spatial access method. The main difference between the proposal and *FastSolvingSet* lies however in the definition of outlier score as the $k$-th neighbour distance, which on one hand enables geometric comparisons that are essential for the pruning strategy, on the other is less robust than the sum of the distances from the $k$-nearest neighbours.

The the latter outlier score has been introduced in Angiulli & Pizzuti (2002) together with the *HilOut* algorithm exploiting space-filling curves for a fast computation of the distance-based outliers.

Bay & Schwabacher (2003) abstract from a specific relation between neighbourhood and outlier status, and propose a nested-loop algorithm to compute top-$n$ outliers using a generic outlier score function which must be anti-monotonic with respect to inclusion of sets of neighbour distances. Objects are loaded into memory in blocks, and for each block the dataset is scanned once to compute the $k$-nearest neighbours of the block's objects. At the end of a scan, the current list of the top-$n$ outliers and a *cutoff* threshold set to the lowest outlier score among the top-$n$ ones are updated. Although worst-case complexity is quadratic, pruning of objects having a score below the lower bound in the current top-$n$ outlier set achieves a near linear performance in practice on randomized datasets. In the experiments, outlier status is defined as membership in the set of the objects having the top-$n$ average $k$ nearest neighbour distances. This work describes a general nested-loop outlier detection framework in which block loading is analogous to candidate selection in *FastSolvingSet*, which however ensures the selection of the objects with the highest provisional scores. Therefore, the cutoff score increases more rapidly in *FastSolvingSet*, enabling a more aggressive pruning. Another difference is that the reported execution times have been obtained after a randomization preprocessing step.

Ghoting et al. (2008) presents the two-phase algorithm RBRP (Recursive Binning and Re-Projection). RBRP adopts the distance to the farthest neighbour as the score of outlier strength and assume the same outlier status as Ramaswamy et al. (2000). In the first phase it partitions the data into bins

in such a way that close objects are likely to be in the same bin and partition members have roughly the same size. A fixed number of k-means steps is executed, and the resulting partition is recursively refined: if a the size of a partition member is smaller than a threshold, the member is sorted by principal component; otherwise, the algorithm is recursively called on the member. In the second phase, one bin at a time is searched for outliers; after each bin is processed the list of the top-$n$ outliers is updated, and a cutoff $c$ is set to the smallest $k$-th neighbour distance in the list. In each bin, for each object a sequential search for $k$ approximate nearest neighbours is performed. The search scans the current bin first, then the remaining bins in order of proximity, until either the $k$-th neighbour distance is smaller than $c$, or all the bins have been scanned. The best and average case complexity of the first phase is $O(N \log N)$, whereas the worst case complexity of both phases is $O(N^2)$. However, in practice RBRP finds approximate neighbours quickly because it processes objects by closeness, and thus can prune the search space earlier. In comparison to Bay & Schwabacher (2003), RBRP execution time is experimentally shown to be smaller by a factor ranging from a few units to an order of magnitude. There are marked differences between this work and ours. The adopted definition of outlier score is the $k$-th neighbor distance instead of the sum of the distances from the $k$-nearest neighbours. The first phase with its final ordering of objects along the principal component axis has no analogue in *FastSolvingSet*, whereas the nested loop in the second phase of Ghoting et al. (2008) is structured essentially the same way as the one in Bay & Schwabacher (2003), thus omitting to create or update the neighborhoods of the visited objects in the inner loop for comparing them to the cutoff as *FastSolvingSet* does.

Vu & Gopalkrishnan (2009) define the outlier score as the cumulative distance to the $k$ nearest neighbours, and outliers as objects having the top-$n$ scores. As in Ghoting et al. (2008), the algorithm, named MIRO (MultI-Rule Outlier), operates in two phases: a k-means clustering phase and a nested-loop phase. With respect to RBRP, k-means operates similarly, but has been optimized and modified to use an additional parameter, which sets the expected

number of objects in each cluster. Additionally, lower and upper bounds for the score of objects and clusters are computed by exploiting the closeness of objects in neighbouring clusters; then, clusters are picked in order of decreasing lower bound, until $n$ objects have been picked; the last lower bound is used as a cutoff to prune all clusters having a smaller upper bound. Finally, the upper bounds of objects in the remaining clusters are individually tested. The nested-loop phase is similar to that of Ghoting et al. (2008), but the initial score cutoff is taken from the first phase, and the triangle inequality is used to further prune objects. Theoretical analysis and experiments show that the quadratic overhead of the first phase due to the search for neighbouring clusters is amortized by savings in she second phase, resulting a linear execution time in practical cases. Overall the algorithm's structure is similar to the one of Ghoting et al. (2008), thus again the main difference is the omission of neighborhood processing for the visited objects in the inner loop over clusters in the final processing. Concerning the added optimizations, notably the computation of an initial lower bound for outlier scores, and the usage of the triangle inequality for pruning objects which cannot be top outliers, the former optimization cannot apply to *FastSolvingSet* owing to the absence of a clustering phase; as to the latter, the triangle inequality is used in *FastSolvingSet* to reduce the number of distance computations, rather than to prune non-outliers. Thus, it is more likely to be effective as dimensionality grows.

Bhaduri et al. (2011) propose an evolution of the algorithm in Bay & Schwabacher (2003), named iOrca, which initially picks an object at random as a reference and computes an in-memory index based on the ordered distances between the reference and the objects in the dataset. Objects are then processed in decreasing order of distances to the reference. When the sum of the distance between the current object and the reference and the $k$-th neighbour distance of the reference is smaller that the cutoff, an application of the triangle inequality allows to prove that the current object and all objects following it in the index order cannot be top-$n$ outliers, thus the algorithm can be stopped immediately. Besides providing an effective early stopping rule, the index also accelerates the

retrieval of $k$-neighbourhoods for not pruned objects: index entries are visited in order of increasing position difference in the index, possibly stopping if the cutoff exceeds the score. Experiments on real datasets show improvements in runtime up to an order of magnitude over Bay & Schwabacher (2003). Processing objects in order of decreasing distance to a random reference object in Bhaduri et al. (2011) and *FastSolvingSet*'s selection of candidates by highest provisional score achieve a similar result, that is, true outliers are more likely to be processed early, thereby increasing the cutoff rapidly. The difference between the techniques lies in the availability of the strength of the potential outlier: it is readily available in Bhaduri et al. (2011) because the index is computed at the beginning, whereas in *FastSolvingSet* the information gradually emerges as more candidates are processed. The index, the triangle inequality and the definition of outlier score as the $k$-th neighbor distances are used in a synergy in Bhaduri et al. (2011) to apply an early stopping rule; in *FastSolvingSet* the same goal is achieved by the removal of all objects from the active set.

Summarizing, all approaches, except the one in the seminal paper (Knorr & Ng, 1998), search for top-$n$ outliers and exploit a basic pruning rule known as Approximate Nearest Neighbour Search, or ANNS (Angiulli & Pizzuti, 2002; Bay & Schwabacher, 2003; Angiulli et al., 2006; Orair et al., 2010): an object is discarded if its current approximate neighbourhood implies a score upper bound that is lower than the cutoff worst known score. Additional strategies prune the search space during neighbour search, or try to improve the effectiveness of ANNS, either by ranking candidate neighbours, or by ranking candidate outliers, thus increasing the cutoff faster. The triangle inequality is used by a minority of approaches to date to prune a candidate outlier (and, in Bhaduri et al. (2011), to halt the algorithm). Although all the *SolvingSet*-based algorithms employ ANNS, the specific algorithm of the present work does also employ the reverse triangle inequality to greatly reduce the number of distance computations between candidate outliers and dataset objects, exploiting the already computed distances stored in the neighbourhood lists of objects.

## 3. Algorithms

In this section, first we briefly recall the key concepts of the *SolvingSet*-based approach, which was used in Angiulli et al. (2006) to predict novel outliers, and then we introduce the new algorithm and discussing it from the computational point of view.

### *3.1. Weights and outliers*

In the following, we assume a dataset $D$ of objects is given, which is a finite subset of a certain metric space.

**Definition 3.1 (Outlier weight)** *Given an object $p \in D$, the* weight $w_k(p, D)$ *of $p$ in $D$ is the sum of the distances from $p$ to its $k$ nearest neighbours in $D$.*

**Definition 3.2 (Top $n$ outliers)** *Let $Top$ be a subset of $D$ having size $n$. If there not exist objects $x \in Top$ and $y$ in $(D \setminus Top)$ such that $w_k(y, D) > w_k(x, D)$, then $Top$ is said to be the* set of the top $n$ outliers in $D$. *In such a case, $w^* = \min_{x \in Top} w_k(x, D)$ is said to be the* weight of the top $n$-th outlier, *and the objects in $Top$ are said to be the* top $n$ outliers in $D$.

### *3.2. The SolvingSet-based approach*

Now we recall the notion of solving set and the *SolvingSet* and the *DistributedSolvingSet* algorithms. The description provided in this section is largely taken from Angiulli et al. (2016), which is also a descendant, along a different research thread, of Angiulli et al. (2006).

**Definition 3.3 (Outlier Detection Solving Set)** *An outlier detection solving set $S$ is a subset $S$ of $D$ such that, for each $y \in D \setminus S$, it holds that $w_k(y, S) \leq w^*$, where $w^*$ is the weight of the top $n$-th outlier in $D$.*

A solving set $S$ always contains the set $Top$ of the top $n$ outliers in $D$. Furthermore, a solving set can be used to predict novel outliers (Angiulli et al., 2006). Our goal is to compute both a solving set $S$ and the set $Top$.

The *SolvingSet* algorithm (Angiulli et al., 2006) is shown in Figure 1 and its working logic is described below. At each iteration (let us denote by $j$ the generic iteration number), the *SolvingSet* algorithm compares all dataset objects with a selected small subset of the overall dataset, called $C_j$ (for *candidate* objects), and stores their $k$ nearest neighbours with respect to the set $C_1 \cup \ldots \cup C_j$. From

**Input**: Dataset $D$, a distance function $dist(\cdot,\cdot)$, integer number $n$ of outliers, integer number $k$ of nearest neighbours, integer number $m$ of candidate points.
**Output**: Solving set of $D$, set of the top-$n$ outliers of $D$.

```
(1)  SolvingSet(D, dist, n, k, m) {
(2)     PointSet SolvSet = new PointSet();
(3)     PointSet C = new PointSet(m);
(4)     MinHeap Top = new MinHeap(n);
(5)     MinHeap NextC = new MinHeap(m);
(6)     for i = 1 to D.length
(7)         D.get(i).NN = new MaxHeap(k);
(8)     C.set(D.RandomSelect(m));
(9)     while C.length ≠ 0 {
(10)       SolvSet.append(C);
(11)       D.drop(C);
(12)       for i = 1 to C.length {
(13)           p = C.get(i);
(14)           for j = 1 to C.length {
(15)               q = C.get(j);
(16)               d = dist(p, q);
(17)               p.NN.updateMin(d);
(18)               if i ≠ j then q.NN.updateMin(d);
(19)           }
(20)       }
(21)       for i = 1 to D.length {
(22)         p = D.get(i);
(23)         for j = 1 to C.length {
(24)           q = C.get(j);
(25)           if max(p.NN.weight(), q.NN.weight()) ≥ Top.min() {
(26)             d = dist(p, q);
(27)             p.NN.updateMin(d);
(28)             q.NN.updateMin(d);
(29)           }
(30)         }
(31)         if p.NN.weight() ≥ Top.min() then NextC.updateMax(p, p.NN.weight());
(32)       }
(33)       for i = 1 to C.length {
(34)           q = C.get(i);
(35)           Top.updateMax(q, q.NN.weight());
(36)       }
(37)       C.set(NextC.get());
(38)    }
(39)    return(⟨SolvSet, Top.getElements()⟩);
(40) }
```

Figure 1: The *SolvingSet* algorithm.

these stored neighbours, an upper bound to the true weight of each data set object can thus be obtained. Moreover, since the candidate objects have been compared with all the dataset objects, their true weights are known. The objects having weight upper bound lower than the $n$-th greatest weight associated with a candidate object, are called *non active* (since these objects cannot belong to the top-$n$ outliers), while the others are called *active*. At the beginning, $C_1$ contains $m$ randomly selected objects from $D$, while, at each subsequent iteration $j$, $C_j$ is built by selecting, among the active objects of the dataset not already inserted in $C_1, \ldots, C_{j-1}$ during the previous iterations, the $m$ objects having the maximum current weight upper bounds. During the computation, if an object becomes non active, then it will not be considered anymore for insertion into the set of candidates, because it cannot be an outlier. As the algorithm processes new objects, more accurate weights are computed and the number of non active objects increases. The algorithm stops when no more objects have to be examined, i.e. when all the objects not yet selected as candidates are non active, and thus $C_j$ becomes empty. The solving set is the union of the sets $C_j$ computed during each iteration.

The *DistributedSolvingSet* algorithm (Angiulli et al., 2013) makes the *SolvingSet* strategy exploitable in parallel/distributed scenarios. It consists of a main cycle executed by a supervisor node, which iteratively schedules the following two tasks: $(i)$ the core computation, which is simultaneously carried out by all the other nodes; $(ii)$ the synchronization of the partial results returned by each node after completing its job. The computation is driven by the estimate of the outlier weight of each data point and of a global lower bound for the weight, below which points are guaranteed to be non–outliers. The above estimates are iteratively refined by considering alternatively local and global information.

It is worth to observe that several mining algorithms deal with distributed dataset by computing local models which are aggregated in a general model as a final step in the supervisor node. The *DistributedSolvingSet* algorithm is different, since it computes the true global model through iterations where only selected global data and all the local data are involved.

The core computation executed at each node consists in the following steps: ($i$) receiving the current solving set objects together with the current lower bound for the weight of the top $n$-th outlier, ($ii$) comparing them with the local objects, ($iii$) extracting a new set of local candidate objects (the objects with the top weights, according to the current estimate) together with the list of local nearest neighbours with respect to the solving set and, finally, ($iv$) determining the number of local active objects, that is the objects having weight not smaller than the current lower bound. The comparison is performed in several distinct cycles, in order to avoid redundant computations. These data are used in the synchronization step, from the supervisor node, to generate a new set of global candidates to be used in the following iteration, and for each of them the true list of distances from the nearest neighbours, to compute the new (increased) lower bound for the weight.

### *3.3. The FastSolvingSet algorithm*

As previously outlined, the goal of the *FastSolvingSet* algorithm is to considerably reduce the number of distance computations required by the *SolvingSet* algorithm to find out a solving set $S$ for a dataset $D$. We note that *SolvingSet* algorithm computes at most $|S \times D|$ distances among the data objects. Why $O(|S \times D|)$, and not exactly $|S \times D|$, distances are computed, is that during the algorithm iterations some distances from the candidate data object $q$ to the data object $p$ are skipped when both $q$ and $p$ have been classified as non active objects. The idea adopted by the *FastSolvingSet* algorithm is that while the algorithm works, the so far learned knowledge can be exploited to pick out distances which would not provide any contribution and hence to avoid calculating them. In other words, when the uselessness of comparing $q$ with $p$ is derivable, the distance separating $q$ from $p$ will be not computed. How this idea is implemented is explained below. During the cycle where each point is compared with each candidate point (lines (21)–(32) in Figure 1), it is possible to skip the comparison of a certain point $p$ with a specific candidate point $q$ if the knowledge of distances of $p$ from the previous candidates (these distances

are stored in the $NN$ heap of $p$ during the previous iterations) suffices to infer the impossibility for $p$ to have $q$ as one of its *knn* set and vice versa. This condition is ensured by the application of a criterion based on the reverse triangle inequality. Specifically, by looking at the aforementioned cycle, we have that at the time the current cycle iteration starts, the following holds:

- let $p \in D$ and $q \in C$ are a generic point of the dataset and a generic candidate point, respectively, such that at least one is an active point, otherwise we would not have any interest in finding their nearest neighbours. We note that while $p$ point may be non-active, the $q$ point is always initially active since it has been chosen as a candidate. However it may became non-active during the loop as a result of the comparisons with other dataset points;
- let $q' \in S$ be the point of $S$ currently closest to $p$ in its $NN$ heap;
- let $d^*$ be the distance of $q$ from its current $k$-th nearest neighbour (that is, the farthest of the current $k$ nearest neighbours); $d^*$ has been computed during the previous iterations and stored in the heap $NN(q)$; we note that $d^*$ is equal to infinite in the case $k$ neighbours of $q$ have not been found yet;
- let $d^{**}$ be the distance of $p$ from its current $k$-th nearest neighbour; $d^{**}$ has been computed during the previous iterations and stored in the heap $NN(p)$;

We can omit the computation of $d(p, q)$ if the following inequalities are true:

$$\begin{cases} d(p,q) > d^* \Longrightarrow p \notin \{k \text{ } nearest \text{ } neighbours \text{ } of \text{ } q\} \\ d(p,q) > d^{**} \Longrightarrow q \notin \{k \text{ } nearest \text{ } neighbours \text{ } of \text{ } p\} \end{cases} \tag{1}$$

By exploiting the reverse triangle inequality to the triangle having the points $p, q, q'$ as vertices (we refer to this specific triangle, for example preferring it to its symmetric triangle having vertices on $p, q, q''$, where $q''$ is the candidate

nearest to $q$, because in this way could use the distance $d(q, q')$ multiple times), we derive that

$$d(p, q) \geq |d(q, q') - d(p, q')| \tag{2}$$

and then the conditions in (1) are true if the following ones hold

$$\begin{cases} |d(q, q') - d(p, q')| > d^* \\ |d(q, q') - d(p, q')| > d^{**} \end{cases} \tag{3}$$

Moreover, it is useful add to the above conditions the test to verify that the involved point ($p$ or $q$) cannot be non-active; in fact, in this case, it is unnecessary to find out the true $k$ nearest neighbours of the point at hand. The conditions in (3) can be rewritten as follows

$$\begin{cases} NOT\ active(q)\ \ OR\ |d(q, q') - d(p, q')| > d^* \\ NOT\ active(p)\ \ OR\ |d(q, q') - d(p, q')| > d^{**} \end{cases} \tag{4}$$

Ultimately, when the conditions in (4) are satisfied, we are can avoid to calculate $d(p, q)$. The above conditions are still correct during the whole execution of the current cycle iteration.

We finally note that the described optimized strategy is also suitable to be used in the case of a distributed scenario, hence defining the *FastDistributed-SolvingSet* algorithm, where the reverse triangle inequality works in the local dataset and guarantees both the effectiveness and efficiency of the computation.

Drawing the conclusions, we remark that the next result immediately follows from the above discussion.

**Theorem 3.4** *The FastSolvingSet (FastDistributedSolvingSet, respectively) algorithm computes the top $n$ outliers of the input dataset $D$.*

### *3.4. Cost analysis*

Not all the distances employed in the *cut-condition* are necessarily known at the time the condition is evaluated. In particular, $d(p, q')$ and $d^{**}$ are stored in the NN heap of $p$, and $d^*$ is stored in the NN heap of $q$. As for $d(q, q')$, despite this distance could have been computed during the previous iterations when $q'$

was included in the set of candidates, it is not necessarily stored in the current NN heaps of $q$ and $q'$.

Thus, the distance $d(q, q')$ is computed the first time it is required to evaluate a *cut-condition*. We note that if we simply replaced the computation of distance $d(p, q)$ with the computation of the distance $d(q, q')$, the above depicted strategy would not provide any temporal advantage. With this aim, whenever a distance $d(q, q')$ is computed, it is then stored in a temporary data structure allowing constant access time (e.g., an hash table) to be reused in other evaluations of the *cut-condition*. Thus, if another dataset point $p'$ has $q'$ as current nearest neighbour in its NN heap, then we can evaluate the *cut-condition* without the need to compute new distances, by retrieving the value $d(q, q')$ in constant time from the above data structure.

Next we analyze the cost of the strategy in terms of distances computed.

Let $m$ be the number of candidates at each iteration ($m \ll |D|$), and let $S_t$ denote the solving set at the beginning of the main $t$-th iteration of the algorithm.

At each iteration, the worst case number of distances $d(q, q')$ is given by $O(m \cdot |S_t|)$, that is the product of the number of current candidates $m$ and the number of distinct old candidates $q'$ stored in the current solving set $S_t$. These distances may allow to save at most $O(m \cdot |D \setminus S_t|)$ distances $d(p, q)$.

By ignoring the $O(|S|^2)$ distances overall computed at the end of the algorithm between each pair of objects belonging to the solving set, the number of distances computed is:

- without the *cut-condition*: $m \cdot |D \setminus S_t|$ at the $t$-th iteration, and $|S| \cdot |D \setminus S|$ in total;
- with the *cut-condition*:
  - in the best case, we have that $q'$ is the same for all points $p$ and that the condition is always satisfied, then only one distance $d(q, q')$ is computed for each candidate $q$ and no $d(p, q)$ is computed: $m$ distances are computed at the $t$-th iteration, and $|S|$ distances in all;

– in the worst case, we have that the $q'$s are all distinct and that the conditions are never satisfied, then all the distances $d(q, q')$ and $d(p, q)$ are computed: $m \cdot |S_t| + m \cdot |D \setminus S_t|$ distances are computed at the $t$th iteration, and $O(|S|^2) + |S| \cdot |D \setminus S|$ distances in all.

Thus, by using the *FastSolvingSet* algorithm the best case in terms of distances saved is $|S| \cdot |D \setminus S| - |S|$, while the worst case in terms of additional distances is $O(|S|^2)$. It can be concluded that the worst case additional number of distances required to incorporate the *cut-condition* in the solving set strategy does not modify the asymptotic cost of the base algorithm, which is already $O(|S|^2) + |S| \cdot |D \setminus S|$. Indeed, in the worst case the optimized algorithm replicates the computation of the $O(|S|^2)$ distances between each pair of solving set objects.

While both the two above depicted bounds are theoretical, in the practice we note that the probability that two or more dataset objects share the same nearest neighbour $q'$ in their NN heaps is not negligible, due to the fact that $S$ is a small subset of the whole dataset. Moreover, the *cut-condition* has in real data a non-negligible probability to be satisfied. We refer to the experimental section for the analysis of the effectiveness of the above strategy.

We also point out that $m \cdot |D \setminus S_t|$ is an upper bound to the number of distances computed during a single main iteration of the algorithm, since the algorithm can avoid the computation of the distance $d(p, q)$ even when both $p$ and $q$ are non-active. In general, the number of such distances is significant.

### *3.5. Strengths and weaknesses of the approach*

The cost analysis of the *FastSolvingSet* algorithm shows that this approach can achieve significant savings in terms of distances to be computed in order to determine the true top outliers. This behavior is particularly advantageous in the presence of large and high-dimensional data. As far as the dataset size is concerned, the achievable absolute time savings grow with the number of objects to be processed. As far as the dataset dimensionality is concerned, the heaviest the distance cost, the greater the advantages offered by this strategy.

Experimental confirmations of these tendencies are provided in the subsequent Section 4.

The above also clarifies the scenarios in which this strategy may result less effective. Indeed, in the presence of a limited number of objects the advantages in terms of computed distances are likely to be counterbalanced by the additional cost associated with cut-conditions computations. A similar behavior is expected if distance computations are cheap.

Moreover, another difficulty may be represented by the intrinsic dimensionality of the data, that is, roughly speaking, the number of statistically independent attributes of the feature space used the represent objects. It is known that as this number grows, the discrimination between distances becomes more and more feeble, also known as the distance concentration phenomenon(Angiulli, 2018). This problem affects all the data analysis approaches based on distances and, hence, also the present one. Nonetheless, it must be said that real data attributes tend to show correlations and, as such, are less subject to the above phenomenon, which instead shows its full presence on synthetically generated independent data.

As for the *FastDistributedSolvingSet* algorithm, an important property of the here introduced strategy is that it requires only local per-node evaluations in that it does not introduce any communication costs or additional calculations in the supervisor node. Since the basic distributed strategy has both a modest communication cost and a limited computation for managing the local results, a key point of the *FastDistributedSolvingSet* is that it is able to efficiently process natively distributed data.

## 4. Experiments

In this section, we present the experiments performed by using the *FastSolvingSet* strategy. The section is organized as follows: Section 4.1 describes the experimental setting and the datasets employed; Section 4.2 and Section 4.3 show the comparison of the sequential and distributed algorithms, respectively. Finally Section 4.4 discusses the performances of the optimized approach when

datasets with increasing data objects number and dimensionality need to be handled.

### *4.1. Experimental setting and datasets*

To outline the effectiveness of the proposed approach, we evaluated the performance of the algorithms through several experiments on large datasets.

In order to guarantee a great level of generality, the algorithm is written in Java and supports communication through the Java libraries implementing the *TCP sockets*. Further, we conducted our experimentation using a normal-profile hardware platform; more precisely, we used 20 workstations, each equipped with an Intel(R) Xeon(R) CPU E5-2670 2.60GHz and 4GB of RAM, interconnected by an Ethernet network with a nominal rate of 100 Mbit/s.

For the centralized experiments, we mainly considered the following five datasets:

- *G3d* is synthetic and contains 500,000 3D real vectors obtained by the union of the objects of three 3-d normal distributions having different mean vectors and the unit matrix as covariance matrix;
- *Covtype* includes the quantitative attributes of the real data set *Covertype* available at the Machine Learning Repository of UCI (Asuncion & Newman, 2007); it consists of 581,012 instances of 10 attributes;
- *G2d* is a synthetic collection of 1,000,000 vectors generated from a 2-d normal distribution having the origin as mean vector and the unit matrix as covariance matrix;
- *Poker* is obtained from the real dataset *PokerHands*, available at UCI repository, by removing the class label; then *Poker* consists of 1,000,000 instances of 10 attributes;
- *2Mass* contains data from the NASA/IPAC Infrared Science Archive[1] (IRSA). Specifically, the dataset is composed of 1,623,376 instances ob-

[1] See `http://irsa.ipac.caltech.edu/`.

tained from the database *2MASS Survey Atlas Image Info* of the 2MASS Survey Scan Working Databases catalog. Each instance consists of three quantitative attributes associated with JHK filters.

For the distributed runs, we used the datasets obtained by randomly partitioning the above centralized datasets. Furthermore, we also built and exploited some variations of the previous datasets in order to highlight the characteristics of the algorithms. In particular, these variations have been obtained by increasing the dimensions of *Covtype* and *G2d*, or by increasing the number of objects of *G2d*.

In the sequel, if not otherwise stated, the values for the detection task parameters, are $n = 10$, $k = 10$, and $m = 100$. We considered also other combinations of values for the above parameters. These additional results are not included in the paper, since they are similar to the reported experiments and they were used to assess the expected behavior of the algorithms.

For the sake of brevity, in the sequel we shorten algorithm names as follows: *SolvingSet* as SS, *FastSolvingSet* as FSS, *DistributedSolvingSet* as DSS, *FastDistributedSolvingSet* as FDSS.

### *4.2. Comparison of sequential versions*

In this section we compare FSS with SS. Table 1 reports the number of the distance computations and the runtime for both the algorithms. It shows that the FSS guarantees notable reduction concerning the number of computed distances.

Let $d_{SS}$ and $d_{FSS}$ denote the number of the distances computed respectively by SS and FSS, then $gain(d) = \frac{d_{SS}-d_{FSS}}{d_{SS}}\%$ denotes the *distance computations gain* of FSS with respect to SS. According to the results in the Table 1, $gain(d)$ ranges from 72.4% (for *G2d*) up to 95.3% (for *Covtype*).

This advantage of FSS allows appreciable time savings with respect to SS.

Let $r_{SS}$ and $r_{FSS}$ are the runtime respectively of SS and FSS, then $gain(r) = \frac{r_{SS}-r_{FSS}}{r_{SS}}\%$ denotes the *runtime gain* of FSS with respect to SS. From the data in the Table 1, $gain(r)$ ranges from 11.9% (for *G2d*) up to 49.3% (for *Covtype*).

| dataset | SS | | FSS | |
|---|---|---|---|---|
| | **distances** | **runtime** | **distances** | **runtime** |
| *G3d* | 0.47 | 35.29 | 0.07 | 29.21 |
| *Covtype* | 6.91 | 449.85 | 0.32 | 227.94 |
| *Poker* | 7.77 | 577.65 | 1.09 | 365.55 |
| *G2d* | 0.42 | 45.46 | 0.12 | 40.07 |
| *2Mass* | 5.65 | 322.69 | 0.36 | 221.12 |

Table 1: Distance computations (in billions) and runtime (in seconds).

Specifically we note that the lowest runtime gain values occur in the smallest datasets (*G2d* and *G3d*).

From the results above reported, there is an appreciable difference between the $gain(d)$ values and the $gain(r)$ ones. This fact mainly depends on two factors: the former is the foreseeable temporal overhead due to the execution of the code that implements the optimization in FSS; the latter is due to how computationally heavy the distance calculation is and hence it obviously depends on the performances of the available hardware but also on the quantitative characteristics of the dataset at hand.

To exemplify the relationship between $gain(r)$ values and the dataset features, we ran specific experiments where a higher distance cost is due to the increasing number of the data attributes. Table 2 shows the runtimes of the algorithms executed on datasets obtained by replicating, respectively, 5, 10, 15, 25, and 50 times all the dimensions of *Covtype*. For example, *Covtype*5* denotes the variant of *Covtype* obtained by replicating 5 times its attributes. The five datasets have a number of attributes ranging from 50 to 500. Furthermore, these datasets retain by construction the same characteristics of *Covtype*, that is they have the same outliers and the *SolvingSet*-based algorithms discover them by computing the same number of distances required for the original dataset (6.91 billions and 0.32 billions, respectively for SS and FSS, as reported in Table 1). Therefore, by observing the results in the above tables, we can state that (i) the increase in runtime comes from the growing of the number of data dimensions since it depends exactly on the greater computational load necessary

| dataset | runtime | |
|---|---|---|
| | SS | FSS |
| *Covtype*5* | 1084.57 | 257.82 |
| *Covtype*10* | 1898.86 | 299.34 |
| *Covtype*15* | 2755.38 | 340.06 |
| *Covtype*25* | 4406.83 | 463.17 |
| *Covtype*50* | 8442.24 | 622.45 |

Table 2: Runtime (in seconds).

for the calculation of the distances and (ii) the larger the computational cost associated with distance calculations, the more appreciable the savings offered by FSS; in fact, the value of $gain(r)$ rises from 49.3% in the case of *Covtype* up to 92.6% for *Covtype*50*.

### *4.3. Comparison of distributed versions*

When the detection task involves huge or distributed datasets and parallel/distributed hardware is available, then the more suitable solution to detect outliers is the use of the distributed algorithm DSS, as it allows significantly shorter computing times with respect to the centralized algorithm SS. Thus, we implemented the optimized distributed version FDSS and ran it to evaluate the impact of optimization on the distributed algorithm. The key outputs of this experimentation are summarized in the following.

Figure 2 and Figure 3 show the results obtained when running the distributed algorithms on a number of computing nodes ranging from 1 up to 20. The plots display the performances of the algorithms compared to that achieved by the centralized base version SS.

As for the distance computations, Figure 2 depicts the ratio between the number of the distances computed by SS and the number of the *equivalent distances* computed by each distributed algorithm (DSS and FDSS). The *equivalent distances* measure plays in the distributed execution the same role of the number of distance computations in a centralized run. More formally, the number of the *equivalent distances* is defined as $\sum_i \max_j\{d_{i,j}\}$, where $d_{i,j}$ is the number of distances computed by node $j$ during the $i$-th iteration. In other words, the

curves showed in Figure 2 represent the *distance computations speedup* of the distributed algorithms. We denote it as $speedup(d)_{DSS}$ and $speedup(d)_{FDSS}$, respectively. The results put in evidence that FDSS guarantees always a better performance than DSS: in fact, DSS (dashed lines) overcomes SS by reaching a distance computation speedup close to linear with respect to the number of nodes, while FDSS (solid lines) gets a larger improvement than DSS for all the datasets. Let $gain(speedup(d)) = \frac{speedup(d)_{FDSS} - speedup(d)_{DSS}}{speedup(d)_{DSS}}\%$ be the *distance computations speedup gain* of FDSS with respect to DSS. Then, by considering the experiments on 20 nodes, the $gain(speedup(d))$ ranges from 273.0% for *Poker* up to 1357.4% for *2Mass*.

Concerning the runtime speedups, the values achieved by DSS and FDSS, respectively denoted by $speedup(r)_{DSS}$ and $speedup(r)_{FDSS}$, are shown in Figure 3. We can note that FDSS outperforms always DSS and the improvement is significant in several cases. Let $gain(speedup(r)) = \frac{speedup(r)_{FDSS} - speedup(r)_{DSS}}{speedup(r)_{DSS}}\%$ be the *runtime speedup gain* of FDSS with respect to DSS. Then, if we look at the execution with 20 nodes, for *2Mass* and *Covtype* the $speedup(r)_{FDSS}$ values are 21.9 and 16.5, corresponding to a $gain(speedup(r))$ values of about 42.5% and 24.4% since the $speedup(r)_{DSS}$ values are 15.4 and 13.3.

The experimental behavior of FDSS confirms the overall efficiency of the optimization even in parallel/distributed domain. This was not a foregone conclusion. First, given that the optimization works locally at each node, it could have a lower quantitative impact, bacause in some nodes it could save less computations. Second, as the execution is distributed, the part of the algorithm that is accelerated by the optimization has a less temporal cost. Neverthless, the above experimental results suggest that other factors supporting the fast version intervene: the first is that the overhead of the optimization is better cushioned with respect to the centralized version; the second is that if it is true that the computational weight of what is accelerated by the optimization decreases in absolute terms, it is also true that it increases in relative terms due to the fact that the distribution of the remaining computation carried out by the algorithm results to be predominant compared to it. Ultimately, the

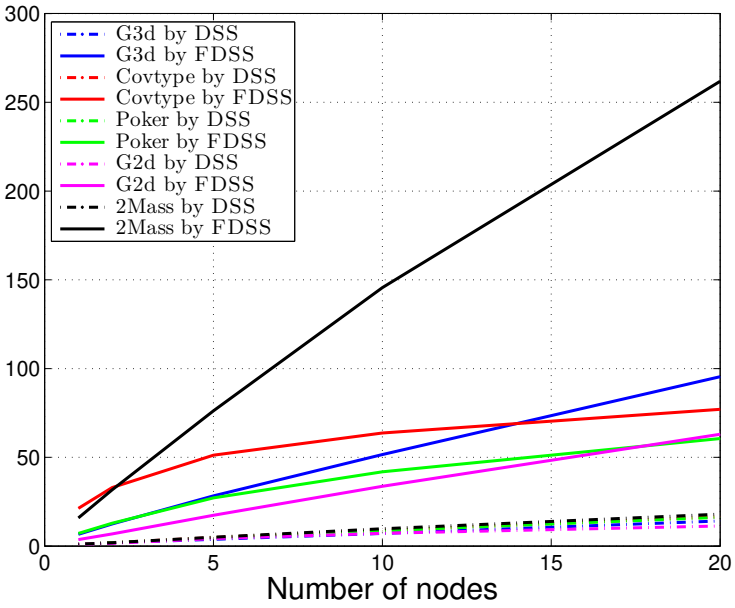



Figure 2: Distance computations speedup.

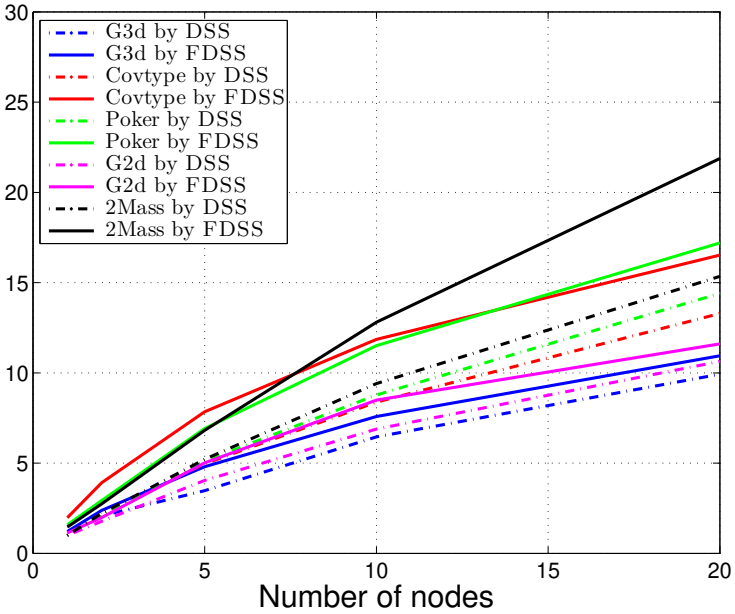



Figure 3: Runtime speedup.

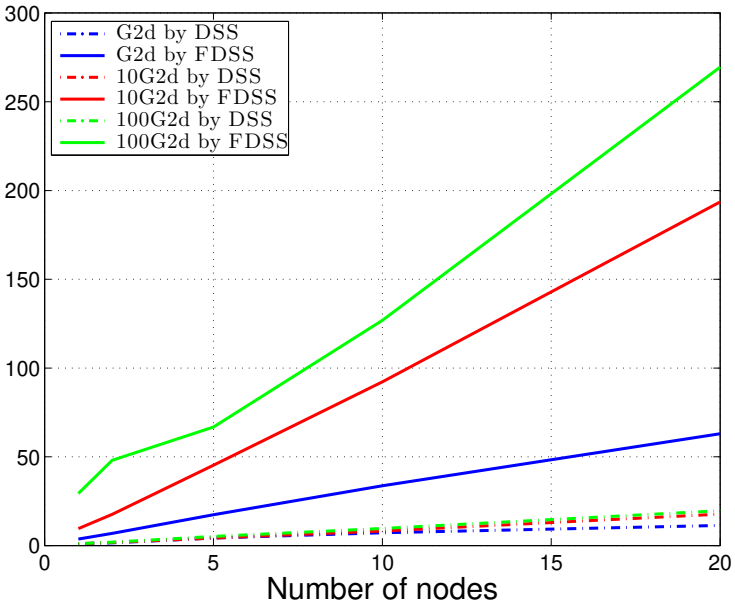



Figure 4: Distance computations speedup.

optimization ensures lower runtimes than the base version.

### *4.4. Analysis for increasing dataset size*

In this section, we show that the optimized algorithm can increase its advantage over the base version when the detection task addresses datasets with a larger number of objects or with objects characterized by a greater number of dimensions. We employed two groups of datasets with increasing number of objects and/or number of dimensions: the first group collects datasets generated in the same manner as *G2d*, whereas the second one gathers the datasets originating from *Covtype* already described in 4.2.

#### *4.4.1. Analysis for increasing dataset objects number*

We used the three datasets *G2d*, *10G2d*, and *100G2d* where the last two have 10 and 100 times the objects as *G2d*, respectively, and ran the algorithms by using up to 20 nodes.

Figure 4 shows the distance computations speedup of DSS and FDSS with respect to SS. As we can see, $speedup(d)_{FDSS}$ has a growing trend larger than

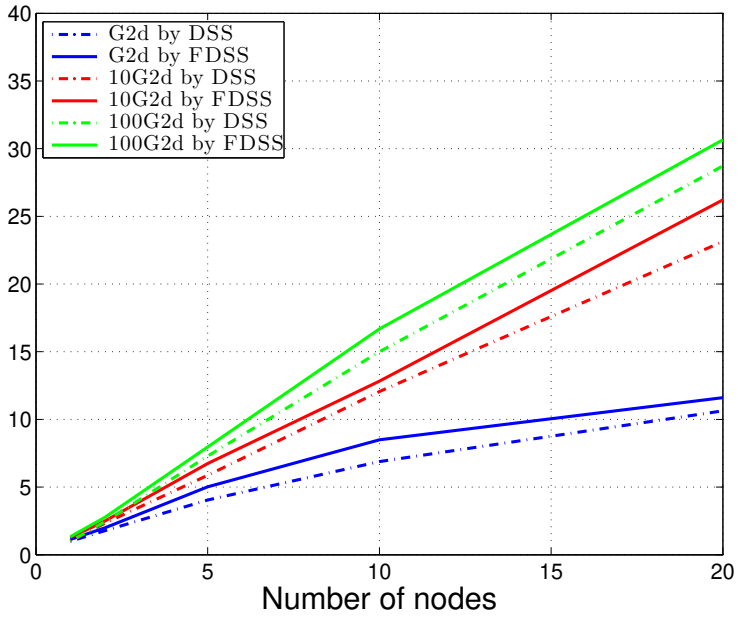



Figure 5: Runtime speedup.

$speedup(d)_{DSS}$ by increasing the number of objects. For example, in the case of 20 nodes, $gain(speedup(d))$ rises of a factor equal to 457.4%, 992.6%, 1272.5% for *G2d*, *10G2d*, and *100G2d*, respectively.

Figure 5 presents the speedup obtained by using the two algorithms. FDSS is always faster than DSS, although the advantage of the former is not vast as for the distance computations. Indeed, for 20 nodes, $gain(speedup(r))$ values are 9.1%, 13.3%, and 6.7% for *G2d*, *10G2d*, and *100G2d*, respectively. This behaviour can be justified because these datasets are composed of two dimensions and therefore the cost of calculating the distances among the objects is quite limited. Furthermore, as far as the two largest datasets are concerned, we observe that the speedup of both algorithms grows faster than any linear one. This trend is a side effect since the centralized execution required the use of an extra management of the data in main memory, so that it has slowed to the point of making superlinear the performances of the distributed algorithms. This fact contributed to cushion the runtime speedup gain powered by FDSS,

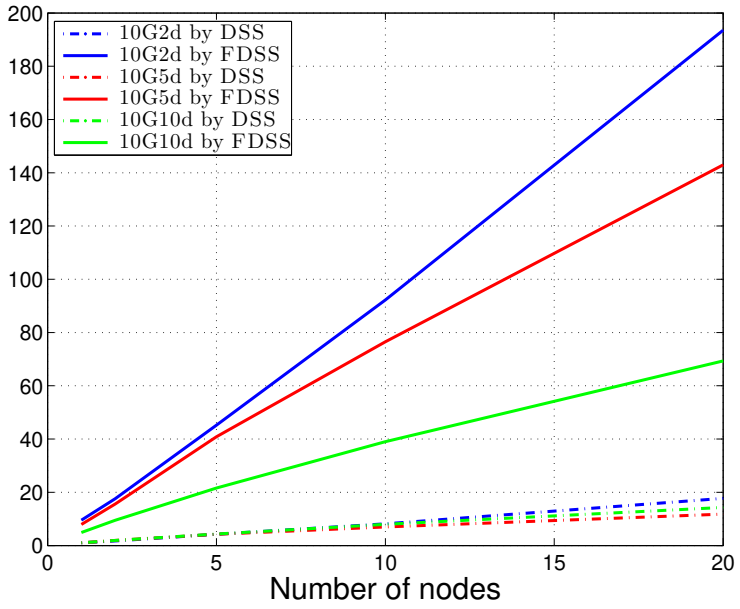



Figure 6: Distance computations speedup.

in particular for *100G2d* for which the impact of the superlinearity is greater.

*4.4.2. Analysis for increasing dataset dimensionality*

In the first experiment, we employed the three datasets *10G2d*, *10G5d*, and *10G10d*, having the same number of objects, but a different number of dimensions.

In Figure 6, the distance computations speedup of the two distributed algorithms with respect to the centralized one is drawn. We note that, even in this case, FDSS allows to notably reduce the computed distances with respect to DSS: for example, when 20 nodes are used, for *10G2d* $speedup(d)_{FDSS}$ is equal to 193.6 whereas $speedup(d)_{DSS}$ is equal to 17.7, for *10G5d* $speedup(d)_{FDSS} = 143.0$ against $speedup(d)_{DSS} = 11.8$, and lastly for *10G10d* $speedup(d)_{FDSS}$ is 69.3 whereas $speedup(d)_{DSS}$ takes 14.3. These results correspond to the following values for $gain(speedup(d))$: $992,6\%$, $1109,2\%$ and $384,4\%$ respectively for *10G2d*, *10G5d*, and *10G10d*. We observe that for the 10-dimensional dataset, the $speedup(d)_{FDSS}$ grows less than the other ones. This tendency is

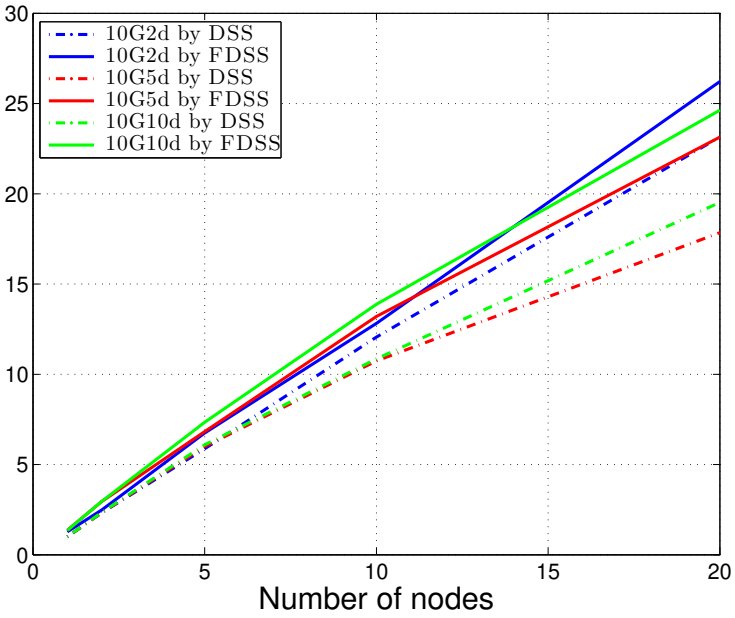



Figure 7: Runtime speedup.

an effect of the well-known *curse of dimensionality problem* (Beyer et al., 1999; François et al., 2007; Angiulli, 2018). In short, by increasing the number of the dimensions, it is more likely the number of times the *cut-condition* is verified to be lower, hence the optimization introduced in FDSS could be less effective. Nevertheless, we put in evidence that also for *10G10d* FDSS shows a distance computations speedup value that improves by 384.4% that of DSS.

From Figure 7, we can observe that for all the examined datasets the $speedup(r)_{FDSS}$ always improves that of DSS; in particular, the greatest improvement occurs for *10G5d*: for example, if we consider the values obtained with 20 nodes, we have that $gain(speedup(r))$ is equal to 13.3%, 29.7%, and 26.2% for *10G2d*, *10G5d*, and *10G10d*, respectively. This behaviour is clarified by taking into account two trends. The first, in favor of FDSS, derives from the fact that, by increasing the number of dimensions, the calculation of the distance between two objects becomes more expensive, and therefore the computational saving allowed by FDSS gets more appreciable. The second, in favor of DSS, derives from the

fact that, as mentioned above, by increasing the number of dimension, the optimization introduced in FDSS reduces its effectiveness. Ultimately, there is a trade-off between these two trends, but if the number of dimensions is such as not to trigger the effects of the curse of dimensionality, then FDSS can guarantee greater efficiency.

In the second experiment, we used the five variants of *Covtype* described in 4.2. This experiment is useful to isolate the impact of the distributed optimized strategy when the increasing of data dimensions does not affect the number of the distance computations necessary to find out the outliers; in fact, as said previously, these datasets present the same outliers and each algorithm detects them computing the same number of distances with to respect the same dataset partition. Practically, in this case, there is not a grow of computational effort due to the curse of dimensionality problem. So the outcome we get just depends on the optimization.

Concerning the computed distances, from Figure 8 we note that the all the solid/dashed lines overlap, that is the obtained values for each algorithm are practically independent of the number of attributes, as it was expected. At the same time we observe that, for each dataset, all the solid lines are above the dashed ones, which implies that the reduction of distance computations of FDSS with respect to DSS is always significant. In the case of 20 nodes, we have that $gain(speedup(d))$ is about 366%.

As for the runtime, the speedups are shown in Figure 9. We can clearly see how the advantage of using FDSS is improved by increasing the cost of the distance calculation due to the increment of the data attributes. For example, we have, for 20 nodes, for *Covtype*5* $speedup(r)_{DSS} = 14.8$ and $speedup(d)_{FDSS} = 30.7$, then $gain(speedup(r)) = 107\%$, whereas for *Covtype*50* $speedup(r)_{DSS} = 16.0$ and $speedup(d)_{FDSS} = 61.4$, then $gain(speedup(r)) = 282.7\%$.

Before concluding, it is also important to underline that if the size of the dataset is such that it requires a mechanism for retrieving data from the disk, then the average cost for calculating a distance would be higher, further favoring the use of FDSS over DSS.

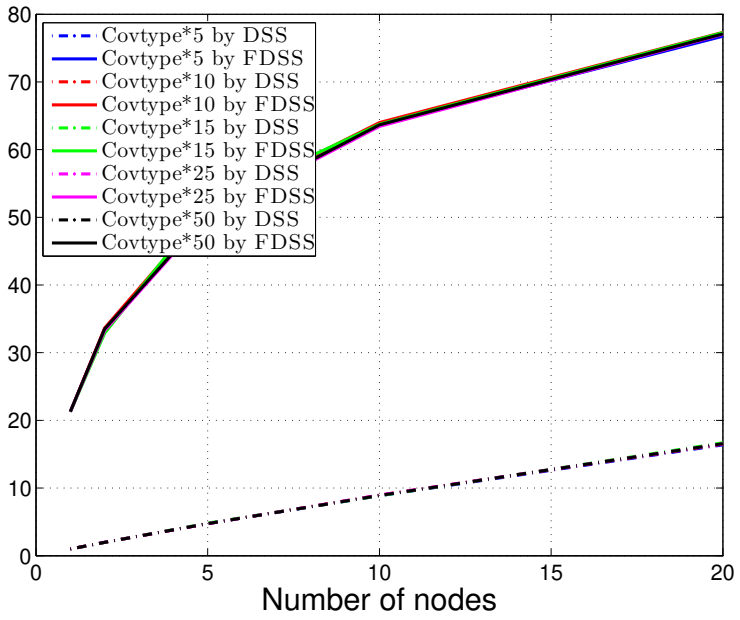



Figure 8: Distance computations speedup.

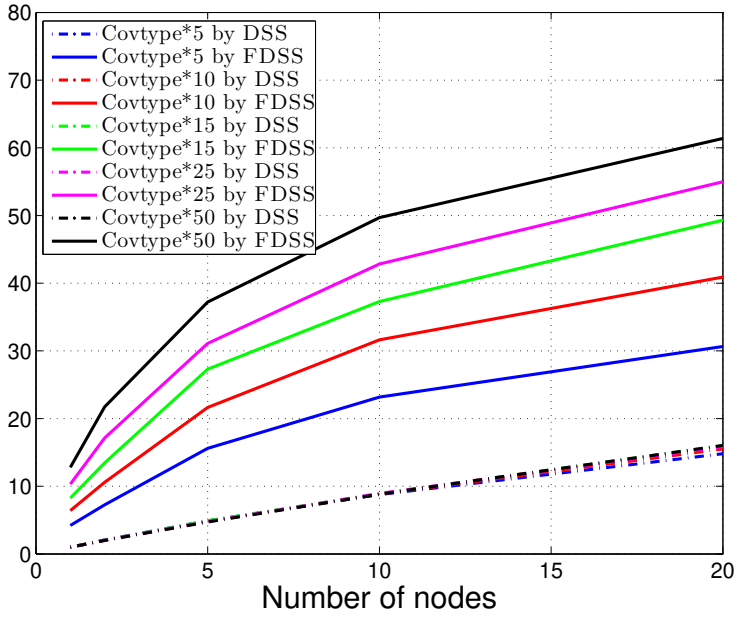



Figure 9: Runtime speedup.

## 5. Discussion and Conclusions

The outlier detection problem is critical for many intelligent applications, such as intrusion or fraud detection, and there is demand for effective solutions in on-line settings and for huge datasets. In the past several researchers, including the authors of this paper, proposed efficient solutions based on the exploitation of massive parallelism. Here we focus on the distance-based outlier detection problem and propose an optimisation which, exploiting the geometrical properties of the problem, substantially reduces the number of distance computations with respect to previous solutions.

Experimental analysis allowed us to obtain confirmation about the expected behaviour of the strategy. Specifically, the reduction in distances computed has been thoroughly noticeable and the temporal advantages amplified by the high-dimensionality of the data.

Experiments conducted on distributed environments showed the effectiveness of the method on natively distributed data, a key requirement for all the emerging intelligent applications dealing with Big Data.

We note that the SolvingSet computation outputs a reduced version of the original dataset that can be used for prediction purposes, that is in the semi-supervised anomaly detection setting. Having a reduced set allows to reduce the spatial cost associated with storing the reference dataset and the temporal cost associated with the process of retrieving the nearest neighbours, thus favouring all the applications where near real-time answer or limited resources represent strict requirements, as in IoT systems.

With reference to future research directions, the expert and intelligent systems applications are expected to need a constant improvement in speed and effectiveness, in order to keep up with increasing amounts and speed of data. The availability of fast, data driven, unsupervised methods for fast finding of outliers could be a success factor. We are exploring several directions of improvement of our techniques. The most relevant are listed next: investigating about a reduction of the gap between the distance computations savings and the asso-

ciated temporal savings; introducing approximate solutions, for a best trade-off between quality of the result and time response; designing on-line strategies, for dealing with incremental or data streaming data; evaluating impact of the distance measure, for the best quality in various application environments.

## References


Angiulli, F. (2018). On the behavior of intrinsically high-dimensional spaces: Distances, direct and reverse nearest neighbors, and hubness. *Journal of Machine Learning Research*, *18*, 1–60.

Angiulli, F., Basta, S., Lodi, S., & Sartori, C. (2013). Distributed strategies for mining outliers in large data sets. *IEEE Transactions on Knowledge and Data Engineering*, *25*, 1520–1532.

Angiulli, F., Basta, S., Lodi, S., & Sartori, C. (2016). GPU strategies for distance-based outlier detection. *IEEE Transactions on Parallel and Distributed Systems*, *27*, 3256–3268.

Angiulli, F., Basta, S., & Pizzuti, C. (2006). Distance-based detection and prediction of outliers. *IEEE Transactions on Knowledge and Data Engineering*, *18*, 145–160.

Angiulli, F., & Fassetti, F. (2009). Dolphin: An efficient algorithm for mining distance-based outliers in very large datasets. *ACM Transactions on Knowledge Discovery from Data*, *3*.

Angiulli, F., & Pizzuti, C. (2002). Fast outlier detection in large high-dimensional data sets. In *Proceedings of the 6th European Conference on Principles and Practice of Knowledge Discovery in Databases* (pp. 15–26). Helsinki, Finland.

Angiulli, F., & Pizzuti, C. (2005). Outlier mining in large high-dimensional data sets. *IEEE Transactions on Knowledge and Data Engineering*, *2*, 203–215.

Asuncion, A., & Newman, D. (2007). UCI machine learning repository. URL: http://archive.ics.uci.edu/ml.

Bay, S. D., & Schwabacher, M. (2003). Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 29–38). New York, NY, USA.

Beyer, K. S., Goldstein, J., Ramakrishnan, R., & Shaft, U. (1999). When is "nearest neighbor" meaningful? In *Proceedings of the 7th International Conference on Database Theory* (pp. 217–235). Jerusalem, Israel.

Bhaduri, K., Matthews, B. L., & Giannella, C. R. (2011). Algorithms for speeding up distance-based outlier detection. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (p. 859–867). San Diego, California, USA.

Chandola, V., Banerjee, A., & Kumar, V. (2009). Anomaly detection: A survey. *ACM Computing Surveys*, *41*, 1–58.

François, D., Wertz, V., & Verleysen, M. (2007). The concentration of fractional distances. *IEEE Transactions on Knowledge and Data Engineering*, *19*, 873–886.

Ghoting, A., Parthasarathy, S., & Otey, M. E. (2008). Fast mining of distance-based outliers in high-dimensional datasets. *Data Mining and Knowledge Discovery*, *16*, 349–364.

Han, J., Kamber, M., & Pei, J. (2011). *Data Mining: Concepts and Techniques*. (3rd ed.). Elsevier.

Knorr, E., & Ng, R. (1998). Algorithms for mining distance-based outliers in large datasets. In *Proceedings of the 24rd International Conference on Very Large Data Bases* (pp. 392–403). New York City, NY, USA.

Orair, G. H., Teixeira, C. H., Meira Jr, W., Wang, Y., & Parthasarathy, S. (2010). Distance-based outlier detection: consolidation and renewed bearing. *Proceedings of the VLDB Endowment*, *3*, 1469–1480.

Ramaswamy, S., Rastogi, R., & Shim, K. (2000). Efficient algorithms for mining outliers from large data sets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 427–438). New York, NY, USA.

Tao, Y., Xiao, X., & Zhou, S. (2006). Mining distance-based outliers from large databases in any metric space. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 394–403). Philadelphia, PA, USA.

Vu, N. H., & Gopalkrishnan, V. (2009). Efficient Pruning Schemes for Distance-Based Outlier Detection. In W. Buntine, M. Grobelnik, D. Mladenić, & J. Shawe-Taylor (Eds.), *Machine Learning and Knowledge Discovery in Databases* (pp. 160–175). Lecture Notes in Computer Science, vol 5782. Springer, Berlin, Heidelberg.

***Highlights (for review)**

HIGHLIGHTS

- The paper introduces the FastSolvingSet algorithm to discover outliers.
- This algorithm computes the distance based outliers with no approximation.
- The experiments outline that a large amount of distance computations is saved.
- FastSolvingSet is suitable to be used in parallel/distributed scenarios.

**LaTeX Source Files and figure files (eps format)**

**Click here to download LaTeX Source Files: source_files.zip**

# CRediT author statement

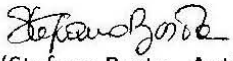
***Fabrizio Angiulli**: Conceptualization, Methodology, Software, Formal analysis, Writing - Original Draft, Visualization.* ***Stefano Basta**: Conceptualization, Methodology, Software, Investigation, Writing - Original Draft, Visualization.* ***Stefano Lodi**: Conceptualization, Methodology, Software, Writing - Original Draft, Visualization.* ***Claudio Sartori**: Conceptualization, Methodology, Software, Writing - Original Draft, Visualization.*

***Conflict of Interest**

**Declaration of interests**

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☒The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:


(Stefano Basta - Author/Authorized Agent For Joint Authors)