Data Mining and Knowledge Discovery (2025) 39:18
https://doi.org/10.1007/s10618-024-01084-1

Efficient outlier detection in numerical and categorical data

Eugênio F. Cabral[1] · Braulio V. Sánchez Vinces[1] · Guilherme D. F. Silva[1] · Jörg Sander[2] · Robson L. F. Cordeiro[3]

Received: 22 September 2023 / Accepted: 28 November 2024

Abstract

How to spot outliers in a large, unlabeled dataset with both numerical and categorical attributes? How to do it in a fast and scalable way? Outlier detection has many applications; it is covered therefore by an extensive literature. The distance-based detectors are the most popular ones. However, they still have two major drawbacks: (a) the intensive neighborhood search that takes hours or even days to complete in large data, and; (b) the inability to process categorical attributes. This paper tackles both problems by presenting HySortOD: a new, fast and scalable detector for numerical and categorical data. Our main focus is the analysis of datasets with many instances, and a low-to-moderate number of attributes. We studied dozens of real, benchmark datasets with up to **one million instances**; HySortOD outperformed **nine competitors** from the state of the art in runtime, being up to **six orders of magnitude faster** in large data, while maintaining high accuracy. Finally, we also performed an extensive experimental evaluation that confirms the ability of our method to obtain high-quality results from both real and synthetic datasets with categorical attributes.

Keywords Outlier detection · Scalability · Numerical and categorical data

Responsible editor: Johannes Fürnkranz.

✉ Robson L. F. Cordeiro
robsonc@andrew.cmu.edu

Braulio V. Sánchez Vinces
braulio.sanchez@usp.br

Jörg Sander
jsander@ualberta.ca

[1] Department of Computer Science, University of São Paulo, Av. Trabalhador São-Carlense, 400, São Carlos, SP 13566, Brazil

[2] Department of Computing Science, University of Alberta, Edmonton, AB T6G 2R3, Canada

[3] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA

Published online: 07 March 2025

1 Introduction

Given a large and unlabeled dataset with instances described by both numerical and categorical attributes, how to detect the outliers? Can it be performed in a fast and scalable manner? Outlier detection is a well-studied data mining task; it is challenging in many aspects, especially because finding an uncommon instance in a large dataset is informally analogous to finding a needle in a haystack. Nevertheless, when these challenges are overcome, outlier detection proves to be valuable in the most diverse areas of human activity, such as in quality control (Jauhri and McDanel 2015), healthCare (Anbarasi and Dhivya 2017), finance (Tripathi et al. 2018) and others (Bindu et al. 2017; Jabez and Muthukumar 2015; Shahid et al. 2015).

A variety of approaches has been developed over the years to meet different application needs (Campos et al. 2016; Aggarwal 2017). The first detectors required a priori knowledge about the data distribution so that statistical tests could be performed to uncover the outliers (Grubbs 1950). These detectors allowed specialists to spot outliers with appropriate accuracy as long as the data followed a known distribution. Unfortunately, most statistical tests are limited to handle only one-dimensional datasets, and it turns out to be very difficult to identify the underlying distribution of the multi-dimensional datasets used in most real-world applications (Aggarwal 2017).

This limitation was overcame with the seminal detector DB-Out (Knorr and Ng 1998). Its authors introduced a distance-based approach that does not require a priori knowledge about the data distribution. The main idea is to run a range query for each instance to retrieve and count its neighbors, and then compare the count of neighbors with a predefined threshold value. If the count is lower than the threshold, then the instance is deemed to be an outlier. DB-Out inspired the development of many other detectors due to its intuitive interpretation of results and wide applicability in different domains. Examples are *k*NN-Out (Ramaswamy et al. 2000), LOF (Breunig et al. 2000), HilOut (Angiulli and Pizzuti 2002), LOCI (Papadimitriou et al. 2003), ODIN (Hautamäki et al. 2004), and MRPG (Amagata et al. 2021, 2022).

One major drawback of a distance-based approach is the intensive neighborhood search that commonly makes it unsuitable for real-world applications. Many researchers recognize this issue, and propose alternatives to mitigate it. For example, *k*NN-Out uses the k nearest neighbor query to control the Cardinality of the sets of neighbors, and it also avoids unnecessary computation by restricting, via user-defined parameters, the number of outliers to be reported. Alternatively, one may use appropriate spatial data structures, such as R-trees and Quad-trees to speed up the neighborhood search, or employ pruning strategies that are available for distance-based algorithms (Orair et al. 2010). Approximate search algorithms such as those proposed in HilOut and aLOCI (Papadimitriou et al. 2003) can also reduce the cost. Nevertheless, in practice, the existing alternatives are still very expensive in terms of runtime: state-of-the-art detectors require hours or even days to process large datasets (Orair et al. 2010; Goldstein and Uchida 2016; Kirner et al. 2017).

Another relevant drawback is the fact that, to our knowledge, no distance-based detector can process categorical attributes. It limits the generality of the existing methods significantly because attributes of this type abound in real applications (Akoglu et al. 2012; Tang et al. 2013). Of course, one can always apply one-hot encoding (Harris and Harris 2012) to convert each categorical attribute into two or more numerical (binary) ones in a pre-processing step; and then, apply the outlier detection method. However, this approach is impractical in most scenarios due to the high dimensionality of the resulting datasets. One may also note that a few non-distance-based detectors can process categorical data. HDoutliers (Wilkinson 2018), iForest (Liu et al. 2008, 2012), COMPREX (Akoglu et al. 2012), and COD (Tang et al. 2013) are the ones we are aware of. Unfortunately, they suffer from other relevant issues, such as being unable to process numerical data, having poor scalability, reporting results that are hard to interpret, or being restricted to specific scenarios, like the detection of only contextual outliers.

The current situation is therefore summarized as Orair et al. (2010); Goldstein and Uchida (2016); Kirner et al. (2017): (a) the existing distance-based detectors struggle to process large numbers of instances, e.g., 100k instances, and; (b) none of these detectors can leverage the categorical attributes that abound in real-world data. By considering these limitations, one can easily question the practicality of such detectors. This paper tackles the problem with the proposal of HySortOD: a novel, distance-based detector that can process both numerical and categorical data in a fast and scalable way. Here, we argue that efficiency and generality of data type are very desirable aspects to be considered in outlier detection. Our main contributions are:

C1 – **Efficiency**: We propose HySortOD – a novel algorithm that efficiently identifies outliers using a new hypercube-ordering-and-searching approach that speeds up the detection task to work at scale;
C2 – **Generality**: We present a new pivot-based strategy that allows our method to tackle both numerical and categorical attributes seamlessly;
C3 – **Benchmark Evaluation**: We study 24 real-world, benchmark datasets with up to 1 million instances, and show that our proposal outperforms 9 competitors of the state of the art in runtime, being up to 6 orders of magnitude faster, while still presenting very accurate results;
C4 – **Case Study**: We investigate the ability of our method to take advantage of categorical attributes, and confirm through an extensive experimental evaluation that these attributes can greatly improve the accuracy of the detection in both real and synthetic data.

Figure 1 summarizes the results obtained for 24 benchmark, real datasets that are popular in the literature. It reports average runtime ranking versus average accuracy ranking for HySortOD (black circles) and 9 competitors of the state of the art. Results for all the datasets studied and for those datasets with only numerical attributes are shown separately on the left side (a) and on the right side (b) of the figure, respectively. An orange cross (at the origin of the plots) marks the position of a fictitious ideal detector that would rank first in both

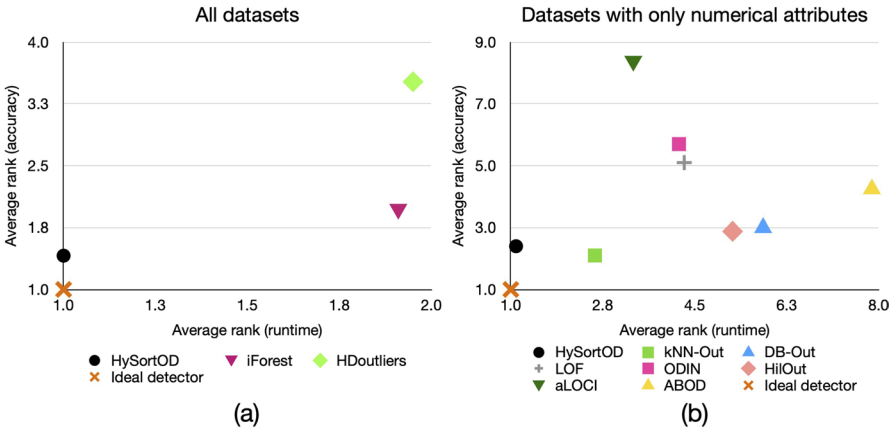


Fig. 1 HySortOD is fast and accurate, even in categorical data: We report runtime ranking versus accuracy ranking for our HySortOD (black circles), and nine state-of-the-art competitors, considering: **(a)** all datasets, and; **(b)** datasets with only numerical attributes. A fictitious ideal detector (orange crosses) would rank first in accuracy and runtime. As shown, our HySortOD is clearly the one that is the closest to the ideal detector in both (a) and (b)

runtime and accuracy. Note that our HySortOD consistently outperformed 9 state-of-the-art competitors, being clearly the one that is the closest to the ideal detector in both (a) and (b). Also, LOF, aLOCI, *k*NN-Out, ODIN, ABOD, DB-Out, and HilOut failed, i.e., could not even be tested, in the datasets with categorical attributes. These results show a significant advantage of our HySortOD over the other detectors. See Sect. 6 for details.

Note Recent and well-known survey papers (Schubert et al. 2014; Campos et al. 2016; Goldstein and Uchida 2016) show that the distance-based detection of outliers works best on data with low-to-moderate dimensionality. Our proposal is no different in this regard, so HySortOD is well suited for datasets with up to nearly 40 attributes. Fortunately, it is well-known that the intrinsic dimensionalities of real datasets are frequently smaller than 40 (Fraideinberze et al. 2016; Traina Junior et al. 2002; Mo and Huang 2012). Thus, if a dataset has more than 40 or so attributes, one can perform distance preserving dimensionality reduction, e.g., by applying PCA, and then detect the outliers with HySortOD. Each categorical attribute must be converted to a numerical one beforehand, using our pivot-based strategy.

Reproducibility For the purpose of reproducibility, all codes, detailed results, parameter values tested and datasets studied in our paper are freely available for download online, at https://github.com/BraulioSanchez/HySortOD.

The rest of this paper is organized as follows. In Sect. 2, we discuss the related work. Our proposal is then introduced in Sects. 3 and 4. In Sects. 5 and 6, we present

the experimental setup and discuss our findings, respectively. Finally, we conclude the paper in Sect. 7.

2 Related work

Detecting outliers in diverse areas of human activity is an important and popular research topic. Many algorithms and renowned surveys (Schubert et al. 2014; Campos et al. 2016; Goldstein and Uchida 2016) on this topic exist in the literature. The distance-based detection of outliers is the most used approach due to its effectiveness, intuitive interpretation of results, and wide applicability in different domains and settings. This approach was introduced with the seminal algorithm DB-Out (Knorr and Ng 1998). DB-Out uses range queries to identify the neighbors of each instance; then, it flags as outliers those instances with few neighbors, that is, fewer than a minimum value defined by the user. Unfortunately, DB-Out requires multiple neighborhood searches that take a long time to complete in datasets of large Cardinality. Due to the cost, its authors also presented a faster, hypercube-based version of the algorithm. A few years latter, LOCI (Papadimitriou et al. 2003) introduced a deviation factor to detect outliers by also leveraging the concept of distance. Like DB-Out, LOCI performs costly range queries. A faster version of LOCI was then presented as an attempt to mitigate the problem, named aLOCI (Papadimitriou et al. 2003). aLOCI starts by creating shifted copies of the dataset; then, it indexes the copies in Quad-trees, and uses the trees as box-counts to estimate the count of neighbors of each instance.

Distance-based detectors may also employ the k-nearest neighbors of each instance. kNN-Out (Ramaswamy et al. 2000) and LOF (Breunig et al. 2000) are probably the first algorithms of this category. Their main advantage is to allow the user to define the number k of neighbors that must be identified per instance, thus providing a better mechanism to control the cost of the search for the neighbors. kNN-Out states that the outlierness of an instance is defined by its distance to the k^{th} nearest neighbor. The authors of kNN-Out also suggest to speed up the detection by reporting only the top-o outliers, instead of a full ranking of the instances. That is, to only identify and report the o instances with the largest distances to their corresponding k^{th} nearest neighbors. LOF introduces a distinct approach. It identifies as outliers those instances with further neighbors than what is expected from their local neighborhoods, where the local neighborhood of an instance is given by its k nearest neighbors. Other algorithms also detect outliers by searching for the k nearest neighbors of each instance. For example, ODIN (Hautamäki et al. 2004) creates a graph of instances (nodes) connected (by edges) to their corresponding k nearest neighbors; then, it flags as outliers those instances whose nodes have in-degrees smaller than a minimum value defined by the user. MRPG (Amagata et al. 2022) also exploits a proximity graph to detect outliers; distinctly from ODIN, it considers three potential definitions for the problem, which are either based on k nearest neighbor queries, or on range queries. Finally, the distance-based detection of outliers have also been investigated in the context of ensembles with the aim of obtaining diversified results

(Kirner et al. 2017), as well as in data streams by means of effective approaches to detect outliers (Tran et al. 2016; Yoon et al. 2018).

The efficiency of a distance-based algorithm typically depends on the use of appropriate data structures, such as R-trees and Quad-trees. It may also depend on restricting the number of outliers to be reported. Approximate neighborhood searches can speed up the detection even further (Orair et al. 2010). For example, hypercube-based approximations have been employed by DB-Out and aLOCI. HilOut (Angiulli and Pizzuti 2002) also performs approximations. It detects outliers by employing Hilbert space filling curves to identify approximate k nearest neighbors for each instance. Nevertheless, HilOut requires several data scans to refine the sets of neighbors, which diminishes the gain of using an approximate strategy. In spite of the aforementioned efforts to speed up the detection, the existing distance-based algorithms still have runtime requirements that are clearly unfeasible for many practical uses. In fact, they commonly require many hours or even days to process a few hundred thousands of instances (Orair et al. 2010; Goldstein and Uchida 2016; Kirner et al. 2017), which may even be seen as a 'light-weight' task in a real-world scenario.

Another relevant drawback is the fact that, to our knowledge, none of the existing distance-based algorithms can process categorical attributes by design, i.e., directly in the way they are defined. Provided that categorical data abound in real applications (Akoglu et al. 2012; Tang et al. 2013), this fact hurts the generality of the distance-based approach as a whole. Of course, one can always employ one-hot encoding (Harris and Harris 2012) to convert categorical attributes to numerical ones in an independent preprocessing step, and then proceed with the detection; or, even use another detector that is not based on distances, like the four ones we are aware of that can handle categorical data: HDoutliers (Wilkinson 2018), iForest (Liu et al. 2008, 2012), COMPREX (Akoglu et al. 2012), and COD (Tang et al. 2013). However, these two approaches have serious limitations: the former suffers from the increased dimensionality of the resulting dataset after attribute conversion; the latter is often unable to process numerical attributes, restricted to specific scenarios (e.g., can only detect contextual outliers), has poor scalability, or reported results that are hard to interpret.

The current situation can therefore be summarized as (Orair et al. 2010; Goldstein and Uchida 2016; Kirner et al. 2017): (a) the existing distance-based detectors struggle to process large numbers of instances (e.g., one hundred thousand instances), and (b) none of these detectors can leverage the categorical attributes that abound in real-world data.

Our work focuses on tackling both of these limitations with a novel algorithm named HySortOD. It uses an efficient hypercube-ordering-and-searching strategy for fast neighborhood search; and, it also takes advantage of one new pivot-based strategy to obtain high-quality results from any dataset, no matter if it includes numerical attributes, categorical ones, or both types of attributes.

Finally, it is worth noting that DB-Out and aLOCI are both bases for our work, as we also employ hypercubes to detect outliers. Our HySortOD improves

upon them by being orders of magnitude faster, and considerably more accurate, as is shown in the experimental evaluation of Sect. 6.

3 Proposed method – basics

This section presents the basic version of our method,[1] which we call here HySortOD$_0$. It is a distance-based algorithm capable of efficiently and effectively detecting outliers in datasets with only numerical attributes. We start with HySortOD$_0$ for clarity of description; in later sections, we provide the additional improvements that lead to our solution for also handling categorical attributes, and finally to our new proposed method HySortOD.

3.1 General idea

HySortOD$_0$ is based on the well-accepted assumption that instances with few neighbors are likely outlying instances, and their closest instances may also be outliers. We consider that the dataset is represented in a multi-dimensional space, and that a distance function is used to distinguish its instances. In this setting, the *outlierness score* of an instance indicates the likelihood of it being an outlier. This concept is formalized in Definitions 1 and 2.

Definition 1 (Range Query) A range query is expressed by a function `Range( )` that returns all instances of a dataset X whose distances to a given query center instance x are within a radius r, according to a distance function $f(\,)$. Formally, it is:

$$\texttt{Range}(X, x, r) = \{x' \;\; x' \in X \;\wedge\; f(x', x) \leq r\}$$

Definition 2 (Outlierness Score) The outlierness score of an instance x in a dataset X is given by the number of neighbors of x within radius r normalized by the maximum number of neighbors existing for any other instance in X. Formally, it is:

$$\texttt{Outlierness}(X, x, r) = 1 - \frac{|\,\texttt{Range}(X, x, r)\,|}{\texttt{Max}(\,\{|\,\texttt{Range}(X, x', r)\,| \;\; : \;\; x' \in X\}\,)}$$

These definitions describe the exact solution for the outlier detection problem that we investigate. Our proposed method HySortOD$_0$ provides an efficient and effective, approximate solution for this problem. In a nutshell, HySortOD$_0$ has four sequential phases. The first phase creates an array of bounded regions, known as hypercubes, to store counts of instances that lie within each region. Next, a novel hypercube-ordering approach is used to organize the hypercubes linearly in an array such that that neighboring hypercubes in the feature space are also close to each other in the linear array. Then, our new neighborhood-search procedure is performed for each hypercube to compute its neighborhood density, that is, the number of instances that lie within the neighborhood. Once the densities are

[1] This basic version was called HySortOD in Cabral and Cordeiro (2020), an earlier version of this work. Here, we use the name HySortOD$_0$ for clarity.

calculated, the last phase reports an outlierness score for each hypercube. Every instance that lies within a hypercube receives the same score – as is common to virtually all hypercube-based algorithms found in the literature. The next sections detail our proposal.

3.2 Creating the hypercubes

The hypercubes are essentially bounded regions of the feature space that contain at least one instance. Without loss of generality, we assume that the dataset $X = \{x_1, x_2, \ldots, x_m\}$ has m normalized instances, where each instance $x_p =< x_{p,1}, x_{p,2}, \ldots, x_{m,d} >$ is within the unit hypercube $[0,\ 1]^d$ with d being the data dimensionality. The bin parameter $b \in \mathbb{N}_{>1}$ represents the number of equi-length partitions to be considered in each dimension, and dictates the hypercube granularity. The length of a hypercube is given by $l = \frac{1}{b}$, which means that the larger the value of b, the smaller the hypercubes are.

As shown in Algorithm 1, we represent the hypercubes in one array $H =< h_1, h_2, \ldots, h_n >$. Each hypercube $h_i =< h_{i,1}, h_{i,2}, \ldots, h_{i,d} >$ is itself an array that stores d coordinates $h_{i,j} \in \{0, 1, \ldots, b-1\}$. Note that H could also be understood as a $n \times d$ matrix, but we decided to use the aforementioned notation because it allows us to describe the rest of our proposal in an easier and clearer manner. We map a dataset instance $x_p \in X$ to hypercube coordinates as follows $h(x_p) =< \lfloor x_{p,1}/l \rfloor, \lfloor x_{p,2}/l \rfloor, \ldots, \lfloor x_{p,d}/l \rfloor >$. Each instance lies within the boundaries of only one hypercube – see Line 2 of Algorithm 1 – and we denote as c_i the count of instances that lie within a hypercube h_i – see Lines 5 and 7. The counts are stored in an array $C =< c_1, c_2, \ldots, c_n >$. The hypercubes are then used as box-counts to estimate their neighborhood densities, as described later in Sect. 3.4.

Algorithm 1 `Create_hypercubes( )`

Input Dataset X; number of *bins* b.
Output Hypercubes H; Counts C.

```
1: for each x_p ∈ X do
2:     Let h_i be the hypercube that x_p is within;
3:     if h_i is an uninitialized hypercube then
4:         Create a new hypercube h_i in H;
5:         c_i ← 0;
6:     end if
7:     c_i ← c_i + 1;
8: end for
9: return H, C;
```

Intuitively, one can see this process as overlaying a *grid* onto the feature space, discretizing and grouping instances. Figure 2a illustrates the mapping process of dataset instances to hypercube coordinates in a 2-dimensional toy dataset; we consider $b = 4$. The grid has therefore up to 16 cells[2]. Note that empty hypercubes are

[2] We use the word *cell* to refer to 2-dimensional hypercubes.

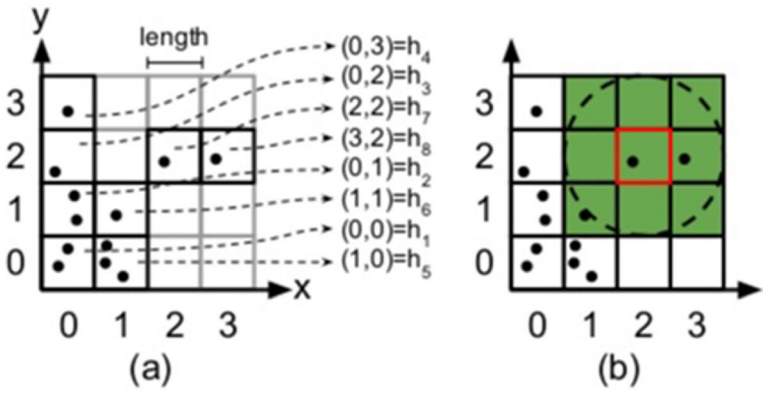


Fig. 2 **(a)** A *grid* using $b = 4$ and 8 *cells* with their coordinates. *Cells* in gray are not stored nor processed. **(b)** The immediate neighbors of *cell* (2, 2) are highlighted in green

not represented in memory nor processed, so the maximum number of hypercubes is always limited by the data Cardinality.

3.3 Sorting the hypercubes

Once the array H of hypercubes is created, the hypercubes are sorted considering their coordinates so that neighboring hypercubes with regard to the feature space are also close to each other in the array. In practice, it is a usual lexicographic ordering procedure that considers d-dimensional hypercube coordinates in numerical order. For example, in Fig. 2a, the hypercubes $H = <(0,3), (0,2), (2,2), (3,2), (0,1), (1,1), (0,0), (1,0)>$ are sorted to $<(0,0), (0,1), (0,2), (0,3), (1,0), (1,1), (2,2), (3,2)>$. This is a simple, yet powerful strategy that can mitigate the costs of a naive neighborhood search by limiting the search space with respect to the neighborhood definition. Our sorting strategy also does not require any advanced indexing structure yet still allows an efficient approach in terms of space complexity. The following section presents its advantages for neighborhood search.

3.4 Searching the neighborhood

The goal of this phase is to efficiently count the number of instances in the neighborhood of each hypercube as a way to measure its neighborhood density. We perform approximate range queries using hypercubes and their coordinates rather than dataset instances. Therefore, for each hypercube $h_i \in H$, we identify its immediate neighbors based on Definition 3.

Definition 3 (Hypercube Neighborhood) Given the hypercube array H and a hypercube h_i of interest, the hypercube neighborhood $\mathtt{N}(h_i)$ includes all hypercubes with

coordinates that are at no more than 1 unit of distance away from the coordinates of h_i. That is: $\mathbb{N}(h_i) = \{h_{i'} \ : \ | \ h_{i',j} - h_{i,j} \ | \leq 1, \ \forall 1 \leq j \leq d; \forall h_{i'} \in H\}$, where $h_{i',j}$ and $h_{i,j}$ are respectively the coordinates of hypercubes $h_{i'}$ and h_i in dimension j.

In Definition 3, the neighboring hypercubes are always at no more than 1 unit of distance away from the hypercube of interest because we assume that a suitable neighborhood radius has been taken into account by the user via parameter b. This definition can be seen as an approximation of a range query (Definition 1) of radius $r = \frac{3l}{2}$ (or diameter $3l$) with the query center x being the centroid of h_i, but the maximum distance between instances within the neighborhood of h_i is actually $3l\sqrt{d}$. Since $l = \frac{1}{b}$, the relationship between r and b is $r = \frac{3}{2b}$. Figure 2b shows the search region according to Definitions 1 and 3; they are respectively given by the dashed circle and the green squares.

Definition 4 denotes the neighborhood density w_i of a hypercube h_i based on the number of instances that exist in its immediate neighbors. The densities are stored in an array $W =< w_1, w_2, \ldots, w_n >$.

Definition 4 (Neighborhood Density) Given a hypercube h_i and its neighbors $\mathbb{N}(h_i)$, the neighborhood density w_i is the count of instances that lie in h_i or in one of its neighbors. Formally, we have:

$$w_i = \sum_{h_{i'} \in \mathbb{N}(h_i)} c_{i'}$$

Let us call any hypercube $h_{i'} \in H$ that is not h_i and satisfies the condition $| \ h_{i,1} - h_{i',1} \ | \ \leq 1$ a *prospective neighbor* of h_i. Similarly, we call any prospective neighbor of h_i that satisfies the full condition in Definition 3 an *immediate neighbor* of h_i. Let us illustrate this idea considering the example of the sorted array H from Sect. 3.3 and a search operation centered at cell h_7 shown in Fig. 2. By sequentially scanning the sorted hypercubes in H, we find the immediate neighbors h_6 and h_8, thus, the neighborhood density is the sum of c_6, c_7, and c_8. Note that the neighborhood search and the neighborhood density computation of a hypercube h_i are performed simultaneously. Thanks to our sorting strategy, the prospective neighbors are already clustered together in array H.

Many prospective neighbors could be tested when performing a linear scan over array H, which would increase the runtime. An ideal solution should minimize such tests and yet find all the immediate neighbors. We observe that it is expected that a sequential range of hypercubes exists – beginning and ending position – that share the same coordinate value for the first dimension; then, fixing the first dimension, we expect for the following dimension other sequences sharing values within the range of the previous dimension, and so on until no more coordinate values are shared. Based on this observation, we can map the hypercubes dimension-wise into a tree-based structure, where each level represents a dimension; the child nodes at each level contain the existing coordinate values for the given dimension as well as the starting and ending positions where the values are shared. This structure allows our HySortOD$_0$ to perform

the density computation by simply traversing specific branches of the tree instead of performing a full scan over array H. We describe the construction and search algorithm in the following sections.

3.4.1 Constructing the dimension-wise tree

Algorithm 2 constructs the dimension-wise tree by recursively scanning array H. For each recursion call, a parent node P and a dimension j must be specified. Each node stores three attributes about the mapping: the coordinate value denoted as P_{value}, besides the beginning and the ending positions in H where the coordinate value is the same for all hypercubes at dimension j, denoted as P_{begin} and P_{end}, respectively. A node cannot map more hypercubes than its parent node does. The root node encloses all hypercubes and does not map a coordinate value of any dimension; that is, for the root node we have: $P_{value} = \emptyset$, $P_{begin} = 0$ and $P_{end} = n$. The first recursion call creates a node for each unique coordinate value at the first dimension, and maps the contiguous interval where each coordinate value is the same. In the following recursion calls, for each of the nodes created by the previous calls, the same procedure is performed to map the hypercubes for the corresponding dimension.

We start by defining the two recursion base cases - see Lines 1 and 2 in Algorithm 2: when the current dimension is larger than the total number of dimensions, and; when the number of hypercubes mapped by a node P is smaller than a predefined threshold *MinSplit*. This threshold aims to balance the trade-off between the number of hypercubes to scan during the neighborhood search and the granularity of mapping. Depending on the data distribution, scanning hypercubes might be faster than traversing the tree branches or scanning leaf nodes that map a single hypercube.

Next, we obtain the first position of the mapped sequence, and the coordinate value of the first hypercube at dimension j – see Line 3. The main loop scans over the hypercubes in H from the beginning to the ending positions mapped by the parent node P – see Lines 4 to 13. During the scan, when the coordinate value changes – see Line 5, a child node is created to map the beginning and ending positions where the coordinate value is the same for all hypercubes within the sequence at dimension j – see Lines 6 and 7. Next, we add the created node as a child of the current parent node P – see Line 8. After that, a recursion is called passing the child node to map the coordinate values of the next dimension – see Line 9. Finally, the beginning and the ending positions of the last coordinate value are added to the parent node aside from the main loop – see Lines 14 to 16. The root node containing all sub-trees is then returned.

Algorithm 2 `Construct( )`

Input Hypercubes H; Value *MinSplit*; Parent node P; Dimension j.
Output Root node P containing all sub-trees.

1: **if** $j > d$ **then return**;
2: **if** $P_{end} - P_{begin} < MinSplit$ **then return**;
3: $i \leftarrow P_{begin}$; $value \leftarrow h_{i,j}$;
4: **while** $i \leq P_{end}$ **do**
5: **if** $h_{i,j} > value$ **then**
6: $begin \leftarrow P_{begin}$; $end \leftarrow i - 1$;
7: Create child node mapping from *begin* to *end*;
8: Add child node into P;
9: `Construct(`H, child node, $j + 1$`)`;
10: $begin \leftarrow i$; $value \leftarrow h_{i,j}$;
11: **end if**
12: $i \leftarrow i + 1$;
13: **end while**
14: $end \leftarrow i - 1$;
15: Add child node mapping from *begin* to *end* in P;
16: `Construct(`H, child node, $j + 1$`)`;
17: **return** P;

In Fig. 3, we illustrate how our proposed dimension-wise tree would be constructed for the 2-dimensional dataset shown in Fig. 2. On the left side, there is the root node mapping all existing sorted hypercubes in H, i.e., from positions h_1 to h_8. Note that the root node has no coordinate value associated to it. Then, the dashed lines point to the child nodes that map the positions for each coordinate value of the first dimension (i.e., dimension x). For the second dimension (i.e., dimension y), there

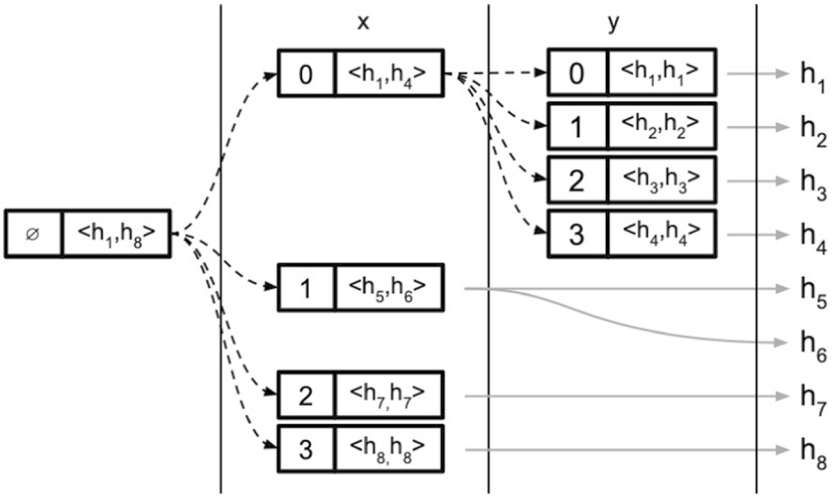


Fig. 3 Dimension-wise tree mapping 8 hypercubes with *MinSplit*=4. The dashed lines indicate the tree hierarchy

are four child nodes for coordinate value 0 from the first dimension because *MinSplit* is set to 4, which means that nodes with the interval that is less than 4 are not split into child nodes – see Line 2 in Algorithm 2. The nodes of the second dimension map only one hypercube, as well as coordinate values 2 and 3 from the first dimension.

The idea of constructing a dimension-wise tree is sensitive to the ordering of the dimensions because some dimensions might provide more information gain than others, and potentially allow one to prune branches during the search step. Thus, this can directly affect the runtime of the neighborhood search. We shall state that we used the original dimension ordering for all datasets studied in this paper. Also, we plan to investigate this point further in future work.

3.4.2 Using the dimension-wise tree

Algorithm 3 shows how to search for all immediate neighbors of a given hypercube, and return its corresponding neighborhood density. Each level of the dimension-wise tree represents a dimension; the nodes in each level store the existing coordinate values with respect to that particular dimension, besides the beginning and ending positions in H. Thus, our strategy is to traverse the branches of the tree, and only scan over hypercubes mapped by the leaf nodes.

The goal is to return the neighborhood density of a given hypercube h_i. Thus, Algorithm 3 starts by assigning the number of instances of hypercube h_i to the total density – see Line 1. When P is a leaf node – see Line 2, then, it proceeds to scan over the mapped interval P_{begin} to P_{end} – see Lines 3 to 5. Within this interval, we test whether the hypercube is an immediate neighbor and increment the total density – see Line 4. When P is not a leaf node, the algorithm recursively traverses through the branches that are 1 unit of distance distant from the given hypercube at each dimension – see Lines 7 to 9, and update the total density. Finally, the total density is reported – see Line 11. In cases where no neighboring hypercube exists for h_i, the total density is c_i.

Algorithm 3 `Neighborhood_density( )`

Input Hypercubes H; Counts C; Position i; Node P; Dimension j.
Output Neighborhood density w_i.

1: $w_i \leftarrow c_i$;
2: **if** P is a leaf **then**
3: **for** $i' \leftarrow P_{begin}$ **to** P_{end} **do**
4: **if** $h_{i,j}$ is immediate of $h_{i',j}$ **then** $w_i \leftarrow w_i + c_{i'}$;
5: **end for**
6: **else**
7: **for each** node mapping the coordinates $h_{i,j} \pm 1$ **do**
8: $w_i \leftarrow w_i +$ `Neighborhood_density`$(H, C,$ node, $i, j+1)$;
9: **end for**
10: **end if**
11: **return** w_i;

3.5 Computing the outlierness scores

The final step is to use the neighborhood density of the hypercubes to assign an outlierness score for every data instance. As it happens in virtually all hypercube-based algorithms found in the literature, we approximate the score of individual instances by reporting the same score for instances that share the same hypercube. Our score is based on Definition 2 and adjusted for the hypercube context, which is the ratio of the hypercube density and the maximum existing density, as shown in Definition 5.

Definition 5 (Hypercube-based Score) Given a hypercube's neighborhood density w_i and the maximum neighborhood density w_{max}, the outlierness score is given by:

$$\texttt{Score}(w_i,\ w_{max}) = 1 - \frac{w_i}{w_{max}}$$

The neighborhood density of a hypercube h_i is w_i, so the score measures the *outlierness* of hypercube h_i based on how dense its neighborhood is when compared to the maximum neighborhood density $w_{max} = \texttt{Max}(W)$, where $W =< w_1, w_2, \ldots w_n >$ is the array with all neighborhood densities. The score values range in the interval [0, 1). High outlierness means scores close to 1, while low outlierness is represented by near-zero scores.

3.6 HySortOD$_0$

Algorithm 4 is the full pseudo-code of our proposed HySortOD$_0$. As we described before, it receives as input the dataset X, an intuitive parameter b that represents the number of *bins* to be created per dimension, and the value *MinSplit* that sets the minimum number of hypercubes a tree node must map to be split into subnodes; consequently, b also defines how close any two instances must be to be considered neighbors. Later, in Sect. 6.3, we analyze the effect of b on the accuracy and the effect of *MinSplit* on the runtime of our method. Additionally, we identify appropriate values for both parameters that allow our proposal to be automatic and yet report fast high-quality results. Algorithm 4 starts by creating the hypercubes H and computing their counts of instances C in Line 1. The hypercubes are then sorted in Line 2 using the procedure described in Sect. 3.3. In Line 3, the hypercubes are mapped into a dimension-wise tree structure. In Lines 4 to 9, the neighborhood densities W are calculated, and the largest one w_{max} is identified. Finally, a data scan on X is performed in Lines 10 to 15 to obtain the outlierness scores O, which are later reported to the user in Line 16.

3.6.1 Time complexity

The complexity for HySortOD$_0$is the sum of the complexity of its individual phases. Time complexity of hypercube creation is $O(m)$ because we only process each instance of dataset X once to output the set of hypercubes of size n, where $n \leq m$. The sorting phase takes $O(dn \log n)$ with traditional sorting algorithms. For the

neighborhood density computation, it is required to make assumptions about the data distribution because the distribution of hypercubes might differ considerably depending on the application. Assuming the *worst-case scenario* where the coordinate values of H are uniformly distributed, we expect for the first dimension, b distinct values and $\frac{n}{b}$ hypercubes sharing the same coordinate value. Based on that, the tree construction takes $O(b\frac{n}{b})$ for the first dimension, and within each node in the first dimension, the child nodes of the second dimension take $b\frac{n}{b^2}$, and so on until dimension d. This process can be represented by the series $\sum_{i=1}^{d} b\frac{n}{b^i}$, where its complexity can be expressed in closed-form as $O(\frac{n(b-b^{1-d})}{b-1})$ or simply $O(3^d n)$. Since at least one instance lies within a hypercube region, the upper bound on the number of immediate neighbors is also limited by n, besides 3^d; that is, the maximum number of immediate neighbors is $s = \texttt{Min}(\{3^d, n\})$. Thus, searching all immediate neighbors for all hypercubes takes $O(sn)$. To report the outlierness of each instance, when efficiently implemented, it takes $O(n)$. Therefore, the final time complexity is $O(sn)$. Note that this analysis relies on the assumption that all instances are uniformly distributed in space, which is the worst-case scenario for *any* outlier detector. Fortunately, this scenario is not expected to occur in practice, because real data tend to be skewed (Faloutsos and Kamel 1994), not uniform. In our experiments, the runtime results have a much lower upper bound, thus further corroborating the fact that the uniform distribution rarely occurs.

The next section provides the additional improvements, that lead to our solution for also handling categorical attributes, and finally to our proposed algorithm HySortOD.

Algorithm 4 HySortOD$_0$( )

Input Dataset X; Number of bins b; Value *MinSplit*.
Output Outlierness scores O for instances of X.

1: H, $C \leftarrow$ `Create_hypercubes(`X, b`)`; ▷ Alg. 1
2: Sort H and adjust C accordingly;
3: `Construct(`H, *MinSplit*, *RootNode*, 1 `)`; ▷ Alg 2
4: Create empty density array W;
5: **for** each $h_i \in H$ **do**
6: $w_i \leftarrow$ `Neighborhood_density(`H, C, *RootNode*, $i, 1$`)`; ▷ Alg. 3
7: Insert w_i into array W;
8: **end for**
9: $w_{max} \leftarrow$ `Max(`W`)`; ▷ Largest density value
10: Create empty outlierness array O;
11: **for** each $x_p \in X$ **do**
12: Let h_i be the hypercube that x_p is within;
13: $o_p \leftarrow$ `Score(`w_i, w_{max}`)`; ▷ Definition 5
14: Insert o_p into array O;
15: **end for**
16: **return** O;

4 Proposed method – HySortOD

Here we show how to handle categorical data, and then finally present the new algorithm HySortOD. In a nutshell, our proposal is twofold: (a) we begin by defining a metric space that properly represents the categorical attributes of the input dataset, and; (b) we then transform this metric space into a nearly equivalent vector space with approximate preservation of the distances between the instances. It leads to a novel strategy that allows us to convert categorical attributes into numerical ones automatically, in a data-driven manner. The new attributes are in turn analyzed following the strategy described previously, in Sect. 3. The details of our proposal are presented in the following.

4.1 Representing the categorical data in a metric space

Here we define a metric space that properly represents categorical attributes. To this end, before looking at categorical data itself, we first analyze how Euclidean[3] distances in numerical data are computed. For example, let us consider a fictitious, 3-dimensional dataset $X = \{x_1, x_2, \ldots, x_m\}$ that includes instances $x_1 =< 0,\ 0,\ 0 >$, $x_2 =< 1,\ 0.5,\ 0 >$ and $x_3 =< 1,\ 1,\ 0 >$, where the three attributes are numerical. Without loss of generality, we assume that the entire dataset is normalized within a unit hypercube; thus, we have that $X \subset [0,\ 1]^3$. Note that, when computing the Euclidean distance $f(x_1,\ x_2) = \sqrt{(0-1)^2 + (0-0.5)^2 + (0-0)^2}$ between instances x_1 and x_2, the first attribute contributes maximally to the total distance; the second attribute contributes a medium value to the total distance, and the third attribute has no contribution to the total distance. Specifically, a numerical attribute contributes maximally to a distance value between two instances if and only if one of the instances has value 0 in this attribute, and the other instance has value 1. The smaller the difference between attribute values, the lower is the attribute's contribution to the total distance. The contribution is null when both instances have the same value for the attribute.

Now, let us consider the particular scenario in which every attribute has either a null contribution or a maximal contribution to the distance between two instances. This is the case for the distance $f(x_1,\ x_3) = \sqrt{(0-1)^2 + (0-1)^2 + (0-0)^2} = \sqrt{2}$ between instances x_1 and x_3 in our example. In cases like that, computing the distance is equivalent to counting the attributes in which both instances have distinct values, and then taking the square root of the count. The instances x_1 and x_3 differ in 2 attributes, and hence the distance between them is $\sqrt{2}$. Importantly, this particular scenario is the basis for our proposal to handle categorical attributes, which is presented in the following.

[3] Note that the reasoning presented in this section is applicable to any L^p-norm with trivial adaptation. Consequently, our proposal is not restricted to the use of Euclidean distances.

Categorical attributes are known for the absence of order in their values. It means that any two instances may either have an identical or a distinct value in any given categorical attribute. The attribute's contribution to the distance between the instances is therefore either null or maximal as discussed for the extreme cases that can occur in normalized numerical data. For instance, let us consider another fictitious, 3-dimensional dataset $Y = \{y_1, y_2, \ldots, y_m\}$ that has instances $y_1 =<$ 'a', 'a', 'a' $>$, $y_2 =<$ 'b', 'a', 'a' $>$ and $y_3 =<$ 'c', 'b', 'a' $>$, where the three attributes are categorical. We assume that $Y \subseteq V_1 \times V_2 \times V_3$ with $V_1 = \{$'a', 'b', 'c'$\}$, $V_2 = \{$'a', 'b'$\}$ and $V_3 = \{$'a', 'b'$\}$. Following the previous discussion, one may think of computing the distance between any two instances of Y by counting the attributes in which they differ, and then taking the square root of the count. For example, y_1 and y_2 differ in only 1 attribute; their distance would therefore be $\sqrt{1}$. Similarly, the distance between y_1 and y_3 would be $\sqrt{2}$.

Nevertheless, this strategy is incorrect as it disregards the number of distinct values that each attribute can assume, i.e., the number of categorical values in each attribute. Clearly, the smaller the number of categories of an attribute, the larger the probability that two instances have the same value in that attribute just by chance. As a consequence, attributes with different numbers of categorical values must have different weights in the distance computation. This line of thought leads us to Definition 6. It presents our proposal to compute distances between the instances of a dataset that has only categorical attributes. Note that there is a probability of $\frac{1}{|V_j|^2}$ that any $y_p, y_{p'} \in Y$ share a particular value in the j^{th} attribute by chance; therefore, the complementary probability $1 - \frac{1}{|V_j|^2}$ is the appropriate weight for the attribute.

Definition 6 (Categorical-only Distance) The distance $f_c(y_p,\ y_{p'})$ between any two instances $y_p =< y_{p,1},\ y_{p,2}, \ldots,\ y_{p,d} >$ and $y_{p'} =< y_{p',1},\ y_{p',2}, \ldots,\ y_{p',d} >$ of a d-dimensional dataset $Y = \{y_1, y_2, \ldots, y_m\}$ is given by:

$$f_c(y_p,\ y_{p'}) = \sqrt{\sum_{j=1}^{d} \begin{cases} 0, & \text{if } y_{p,j} = y_{p',j} \\ 1 - \frac{1}{|V_j|^2}, & \text{otherwise} \end{cases}}$$

where $Y \subseteq V_1 \times V_2 \times \ldots V_d$, and each V_j is the set of categorical values for the j^{th} attribute of Y.

Before moving on to converting categorical attributes into numerical ones, we demonstrate that f_c is a metric.

Lemma 1 *The Categorical-only Distance f_c is a metric. That is, for any instances $y_p, y_{p'}, y_{p''}$ of a dataset $Y \subseteq V_1 \times V_2 \times \ldots V_d$, we have that*:

- $f_c(y_p,\ y_{p'}) \geq 0$;

- $f_c(y_p,\ y_{p'}) = 0$ if and only if $y_p = y_{p'}$;
- $f_c(y_p,\ y_{p'}) = f_c(y_{p'},\ y_p)$;
- $f_c(y_p,\ y_{p'}) \leq f_c(y_p,\ y_{p''}) + f_c(y_{p''},\ y_{p'})$ (triangle inequality).

Proof Function f_c can be rewritten as:

$$f_c(y_p,\ y_{p'}) = \sqrt{f_w(y_p,\ y_{p'})}$$

where

$$f_w(y_p,\ y_{p'}) \ = \ \sum_{j=1}^{d} \begin{cases} 0, & \text{if } y_{p,j} = y_{p',j} \\ 1 - \frac{1}{|V_j|^2}, & \text{otherwise} \end{cases}$$

As a consequence, proving that f_c is a metric is equivalent to proving both that: (a) f_w is a metric, and; (b) the square root of a metric is also a metric. To prove the former,[4] we must show that:

1. $f_w(y_p,\ y_{p'}) \geq 0$;
2. $f_w(y_p,\ y_{p'}) = 0$ if and only if $y_p = y_{p'}$;
3. $f_w(y_p,\ y_{p'}) = f_w(y_{p'},\ y_p)$;
4. $f_w(y_p,\ y_{p'}) \leq f_w(y_p,\ y_{p''}) + f_w(y_{p''},\ y_{p'})$ (triangle inequality).

#1, #2 and #3 are obvious. To prove #4, let us first write f_w in terms of a distance function for an individual attribute:

$$f_w(y_p,\ y_{p'}) = \sum_{j=1}^{d} f_a(y_{p,j},\ y_{p',j})$$

where

$$f_a(y_{p,j},\ y_{p',j}) = \begin{cases} 0, & \text{if } y_{p,j} = y_{p',j} \\ 1 - \frac{1}{|V_j|^2}, & \text{otherwise} \end{cases}$$

We first prove #4 for data of dimensionality $d = 1$:

[4] Note that f_w resembles the Hamming distance function (Robinson 2003), which is a well-known metric as demonstrated by Brian Marcus (Marcus 2022). Specifically, f_w can be seem as a "weighted version" of the Hamming distance. We prove that it is a metric using the demonstration of Marcus as a basis.

- *Case* 1: If $y_p = y_{p'}$, then $f_w(y_p, y_{p'}) = 0$, and therefore $f_w(y_p, y_{p'}) \leq f_w(y_p, y_{p''}) + f_w(y_{p''}, y_{p'})$;
- *Case* 2: If $y_p \neq y_{p'}$, then $f_w(y_p, y_{p'}) = 1 - \frac{1}{|V_j|^2}$, and either $y_p \neq y_{p''}$ or $y_{p''} \neq y_{p'}$ is true, or both are true. Therefore, $f_w(y_p, y_{p''}) + f_w(y_{p''}, y_{p'})$ is at least $1 - \frac{1}{|V_j|^2}$. As a consequence, $f_w(y_p, y_{p'}) \leq f_w(y_p, y_{p''}) + f_w(y_{p''}, y_{p'})$. Note that $j = 1$, since $d = 1$.

For general d:

- By the case of $d = 1$, for each j, we can affirm that $f_a(y_{p,j}, y_{p',j}) \leq f_a(y_{p,j}, y_{p'',j}) + f_a(y_{p'',j}, y_{p',j})$;
- As a consequence:

$$\sum_{j=1}^{d} f_a(y_{p,j}, y_{p',j}) \leq \sum_{j=1}^{d} f_a(y_{p,j}, y_{p'',j}) + f_a(y_{p'',j}, y_{p',j})$$

$$\sum_{j=1}^{d} f_a(y_{p,j}, y_{p',j}) \leq \sum_{j=1}^{d} f_a(y_{p,j}, y_{p'',j}) + \sum_{j=1}^{d} f_a(y_{p'',j}, y_{p',j})$$

$$f_w(y_p, y_{p'}) \leq f_w(y_p, y_{p''}) + f_w(y_{p''}, y_{p'})$$

It proves that f_w is a metric. Finally, we must demonstrate that the square root of f_w is also a metric, thus:

1. $\sqrt{f_w(y_p, y_{p'})} \geq 0$;
2. $\sqrt{f_w(y_p, y_{p'})} = 0$ if and only if $y_p = y_{p'}$;
3. $\sqrt{f_w(y_p, y_{p'})} = \sqrt{f_w(y_{p'}, y_p)}$;
4. $\sqrt{f_w(y_p, y_{p'})} \leq \sqrt{f_w(y_p, y_{p''})} + \sqrt{f_w(y_{p''}, y_{p'})}$ (triangle inequality).

#1, #2 and #3 are obvious and #4 can be easily derived from the triangle inequality of the metric f_w: $f_w(y_p, y_{p'}) \leq f_w(y_p, y_{p''}) + f_w(y_{p''}, y_{p'})$. Since all values are positive it holds that: $f_w(y_p, y_{p''}) + f_w(y_{p''}, y_{p'}) \leq f_w(y_p, y_{p''}) + f_w(y_{p''}, y_{p'}) + 2 \cdot \sqrt{f_w(y_p, y_{p''})} \cdot \sqrt{f_w(y_{p''}, y_{p'})} = (\sqrt{f_w(y_p, y_{p''})} + \sqrt{f_w(y_{p''}, y_{p'})})^2$, i.e., we have $f_w(y_p, y_{p'}) \leq (\sqrt{f_w(y_p, y_{p''})} + \sqrt{f_w(y_{p''}, y_{p'})})^2$ $f_w(y_p, y_{p'}) \leq (\sqrt{f_w(y_p, y_{p''})} + \sqrt{f_w(y_{p''}, y_{p'})})^2$. Since both sides are non-negative numbers, taking the square root on both sides yields the triangle inequality: $\sqrt{f_w(y_p, y_{p'})} \leq \sqrt{f_w(y_p, y_{p''})} + \sqrt{f_w(y_{p''}, y_{p'})}$. □

As a consequence, space $\langle V_1 \times V_2 \times \ldots V_d, f_c \rangle$ is metric. It applies to any dataset $Y \subseteq V_1 \times V_2 \times \ldots V_d$ having only categorical data. The next step is to capitalize on this space to convert categorical attributes into numerical ones.

4.2 Converting the metric space into a vector space

Here we show how to transform the metric space $\langle V_1 \times V_2 \times \ldots V_d, f_c \rangle$ into a nearly equivalent vector space with approximate preservation of the distances between the instances. It leads to a new strategy that allows us to obtain numerical attributes from categorical ones automatically, in a data-driven way.

To this end, we leverage ideas introduced with the Omni-family of metric access methods (Traina Junior et al. 2007), and propose to reuse them with a novel purpose: attribute conversion. The Omni access methods were originally developed to speed up the execution of range and kNN queries in a metric dataset. The main idea is to select a small subset of the instances to be pivots and then represent instances of the dataset in a nearly equivalent vector space by means of the distances between the instances and the pivots. Importantly, it is shown that each pivot must be far from all the other pivots; and, there is a minimum number of pivots required to create a vector space that preserves the distances between the instances with a negligible error. The authors demonstrate that this minimum number of pivots is the (correlation) fractal dimension of the dataset. The queries are later executed by taking advantage of the vector data (i.e., the distances to the pivots) to find candidate instances for the result set, which are finally confirmed by computing a few distances in the original space.

Algorithm 5 `Metric_to_vector( )`

Input Dataset Y. ▷ Assumption: $Y \subseteq I \times V_1 \times V_2 \times \ldots V_d$
Output Dataset X. ▷ Result: $X \subset I \times [0,\ 1]^d$

$X \leftarrow \{\ \}$; ▷ Resulting dataset
$U \leftarrow \{\ \}$; ▷ Set of pivots
if $d > 1$ **then** ▷ **Two or more attributes:** one pivot per attribute
  Randomly choose an instance $y_p \in Y$;
  Find the instance $y_{p'} \in Y$ that maximizes $f_c(y_p,\ y_{p'})$;
  Let $y_{p'}$ be the **first pivot** u_1; ▷ Far from a random instance
  $U \leftarrow U \cup \{u_1\}$;
  Find the instance $y_p \in Y$ that maximizes $f_c(u_1,\ y_p)$;
  Let y_p be the **second pivot** u_2; ▷ Far from the first pivot
  $U \leftarrow U \cup \{u_2\}$;
  $Edge \leftarrow f_c(u_1,\ u_2)$; ▷ Desired distance between every pair of pivots
  for $j \leftarrow 3$ **to** d **do** ▷ Find the remaining pivots
    for each $y_p \in Y \setminus U$ **do**
      $\texttt{error}(y_p) \leftarrow \sum_{u_{j'} \in U} \mid Edge - f_c(u_{j'},\ y_p) \mid$
    end for
    Find the instance $y_p \in Y \setminus U$ that minimizes $\texttt{error}(y_p)$;
    $u_j \leftarrow y_p$; ▷ $\mathbf{j^{th}}$ **pivot**; far from the previous pivots
    $U \leftarrow U \cup \{u_j\}$;
  end for
  for each $y_p \in Y$ **do** ▷ Build x_p using distances from y_p to the pivots
    for each $u_j \in U$ **do**
      $x_{p,j} \leftarrow f_c(u_j,\ y_p)$
    end for
    $X \leftarrow X \cup \{x_p\}\ :\ x_p = < id(y_p),\ x_{p,1},\ x_{p,2}, \ldots\ x_{p,d} >$;
  end for
else ▷ **Only one attribute:** one pivot per infrequent value
  $A \leftarrow <a_1, \ldots, a_{|V_1|}>$ with $a_e = <k_e,\ g_e>$ and ▷ Sorted by frequency
    k_e is the $\mathbf{e^{th}}$ **least frequent** value of the attribute of Y, while
    g_e is the corresponding frequency;
  $q \leftarrow \texttt{Cutoff(}\ < g_1, \ldots, g_{|V_1|} >\ \texttt{)}$; ▷ Definitions 7 and 8
  for $e \leftarrow 1$ **to** q **do** ▷ Find the pivots
    Randomly choose an instance $y_p \in Y$ with $y_{p,1} = k_e$;
    $u_e \leftarrow y_p$; ▷ $\mathbf{e^{th}}$ **pivot**; it represents the infrequent value k_e
    $U \leftarrow U \cup \{u_e\}$;
  end for
  for each $y_p \in Y$ **do** ▷ Build x_p using distances from y_p to the pivots
    $x_p \leftarrow < id(y_p),\ x_{p,1} >\ :\ x_{p,1} = \sum_{u_e \in U} f_c(u_e,\ y_p)$;
    $X \leftarrow X \cup \{x_p\}$;
  end for
end if
Normalize X;
return X;

To our knowledge, we are the first ones to leverage these ideas for attribute conversion. Algorithm 5 is the pseudo-code of our proposal. Note that it considers that instances have identifiers that were omitted in the previous sections for clarity. The algorithm receives a dataset $Y \subseteq I \times V_1 \times V_2 \times \ldots V_d$ having d categorical attributes, such that I is the set of identifiers of instances and $V_1, V_2, \ldots V_d$ are the corresponding sets of categorical values. Then, it returns a nearly equivalent, vector dataset $X \subset I \times [0,\ 1]^d$ with approximate preservation of the distances between the instances. Except for the case when Y has a single attribute – which is treated later in this section – we follow the Omni approach and select a small subset of instances $U \subset Y$ to be the pivots of the analysis; see Lines 4-19 in Algorithm 5. Note that we always select d pivots in total; it allows us to generate a new dataset X that has the same dimensionality as the original dataset Y, and also to have a safely large set of pivots, that is, to always have $\mid U \mid$ larger than or equal to the (correlation) fractal dimension of Y. Let us emphasize that the (correlation) fractal dimension measures the number of degrees of freedom of the underlying mechanism that generated the data (Schroeder 1991; Faloutsos and Kamel 1994), and thus, by definition, it cannot be larger than the total number of attributes. Additionally, it is guaranteed that the pivots are far from one another. To this end, the first pivot u_1 is the instance of Y that maximizes the distance to a randomly selected instance $y_p \in Y$; and, each of the remaining pivots u_j is chosen efficiently with the aim of having large distances to the pivots selected previously. Finally, the new dataset X is built from the distances between every instance in Y and each pivot in U; see Lines 20-25.

Now we focus on the particular case when Y has only one attribute. The Omni approach is not suitable in this case because: (a) if we follow the previous ideas and select d pivots – that is, only one pivot – we would not meet the requirement that the pivots must be far from one another, and; (b) if we change the procedure to selecting two or more pivots we would end up increasing the dimensionality of the original data, which is not desirable. Consequently, this case requires a slightly different approach: we build on the Omni ideas and present a novel procedure that allows using more than one pivot yet preserves the data dimensionality. Specifically, we aim at generating a vector dataset X that distinguishes outliers and inliers according to Y. To this end, we propose to have a set of pivots $U \subset Y$ in which each pivot is an instance with one *infrequent* attribute value from Y. Then, we generate each $x_p \in X$ from the *sum of the distances* between instance $y_p \in Y$ and each pivot $u_e \in U$. It helps us distinguish inliers from outliers because the former tend to be far from all pivots (i.e., inliers should have frequent values only), while the latter tend to be close to one of the pivots (i.e., each outlier should have one of the infrequent values). Therefore, if an instance $y_p \in Y$ is an inlier, then it should have a larger sum of distances to the pivots than that of an outlier instance. Note that this reasoning assumes that the original attribute is relevant to the detection of outliers, that is, we suppose that the categorical values in Y allow distinguishing outliers from inliers; otherwise, dataset Y has no useful information, and so does the dataset X generated from it.

To find the infrequent values we create a sorted list $A =$ where each $a_e =$ is a pair containing the e^{th} *least frequent* value k_e of the attribute of Y and the corresponding frequency g_e. Then, we partition the list of frequencies $< g_1, \ldots, g_{|V_1|} >$ to create two

sub-lists that are as homogeneous as possible. Specifically, as shown in Lines 27-28 of Algorithm 5, we find the *Cutoff q* that maximizes the homogeneity of the sub-lists $< g_1, \ldots, g_q >$ and $< g_{q+1}, \ldots, g_{|V_1|} >$, so as to best separate the small frequencies $g_1, \ldots, g_q$ from the large frequencies $g_{q+1}, \ldots, g_{|V_1|}$. The infrequent values $k_1, \ldots, k_q$ are then found without depending on any input from the user.

To obtain the Cutoff q, based on the principle of Minimum Description Length, we analyze sub-lists $< g_1, \ldots, g_e >$ and $< g_{e+1}, \ldots, g_{|V_1|} >$ for all possible partitioning positions $e \in \{1, \ldots, \mid V_1 \mid -1\}$ as shown in Definitions 7 and 8. The idea is to compress each possible sub-list, representing it by its Cardinality, its average, and the differences of each of its elements to the average.[5] A highly homogeneous sub-list allows good compression, as the differences to the average are small, and small numbers need fewer bits to be represented than larger numbers. The Cutoff q is therefore identified as the partitioning position that minimizes the cost of compressing the corresponding sub-lists.

Definition 7 (Cost of Compression) The cost of compression of a sorted, non-empty list of positive integers $< g_1, \ldots, g_t >$ is defined as

$$\texttt{Cost}(< g_1, \ldots, g_t >) = \log^*(t) + \log^*\left(\left\lceil \bar{g} \right\rceil\right) + \sum_{e=1}^{t} \left(\log^*\left(1 + \left\lceil \mid g_e - \bar{g} \mid \right\rceil\right) + \log^*(2) \right),$$

where $\bar{g} = \frac{\sum_{e=1}^{t} g_e}{t}$, and $\log^*$ is the universal code length for positive integers[6].

Definition 8 (Cutoff) The function `Cutoff( )` is defined as

$$\texttt{Cutoff}(< g_1, \ldots, g_{|V_1|} >) = e \in \{1, \ldots, \mid V_1 \mid -1\} \text{ such that } e \text{ minimizes } \texttt{Cost}(< g_1, \ldots, g_e >) + \texttt{Cost}(< g_{e+1}, \ldots, g_{|V_1|} >).$$

Once the infrequent values are known, we create the set of pivots U in Lines 29–33 of Algorithm 5. Each pivot is an instance of Y that has one of the infrequent values. Then, in Lines 34–37, the new dataset X is built from the sum of the distances between every instance in Y and each pivot in U. Finally, dataset X is normalized in Line 39, and then it is returned in Line 40.

[5] Note in Definition 7 that we add one to each difference to the average whose code length $\log^*$ is required. It allows accounting for zeros. The cost of registering if each difference to the average is positive or negative is given by $\log^*(2)$.

[6] It can be demonstrated that $\log^*(g) \approx \log_2(g) + \log_2\left(\log_2(g)\right) + \ldots$, where $g \in \mathbb{N}_{\geq 1}$ and only the positive terms of the equation are retained (Rissanen 1983; Chakrabarti et al. 2004). This is the optimal length, if we do not know the range of values for g beforehand.

4.3 HySortOD

Now, we finally present HySortOD. Distinctly from the basic method of Sect. 3, the proposed method receives as input a dataset $Z = \{z_1, \ldots, z_m\}$ that may have numerical attributes, categorical attributes, or both types of attributes. In our notation, every instance z_p is defined as $z_p = < id(z_p), z_{p,1}, z_{p,2}, \ldots z_{p,d} >$, in which $id(z_p)$ is the identifier of the instance, $z_{p,j}$ is the instance's value for the j^{th} attribute and d is the data dimensionality. For easy of presentation – and also, without loss of generality – we assume an ordering of the attributes where the numerical ones come first; and each numerical attribute is normalized within [0, 1]. Our working scenario can therefore be written as: $Z \subset I \times [0,\ 1]^{d'} \times V_1 \times V_2 \times \ldots V_{d-d'}$, where I is the set of identifiers of instances, d' is the number of numerical attributes, and $V_1, V_2, \ldots V_{d-d'}$ are the sets of categorical values of each of the $d - d'$ categorical attributes.

Algorithm 6 is the pseudo-code of HySortOD. We begin by identifying whether or not all attributes in Z are numerical. If so, we simply use HySortOD$_0$ to process the data; see Line 2. Otherwise, we isolate the numerical values into a dataset X' and the categorical ones into a dataset Y. Then, we convert Y into a nearly equivalent, vector dataset X''. Later, we run a join between X' and X'' to match the instances by their identifiers, thus generating a single dataset X. Afterward, we use HySortOD$_0$ to process X. See Lines 3-15 for the details. Finally, in Line 16, we return the resulting scores O.

Algorithm 6 HySortOD ( )

Input Dataset Z; ▷ Assumption: $Z \subset I \times [0,\ 1]^{d'} \times V_1 \times V_2 \times \ldots V_{d-d'}$
Note that d' is the number of numerical attributes.
Number of bins b;
Value *MinSplit*.
Output Outlierness scores O for instances of Z.

1: **if** $d = d'$ **then** ▷ All attributes are numerical
2: $O \leftarrow$ `HySortOD`$_0(X \leftarrow Z,\ b,\ MinSplit)$; ▷ Detect outliers using Alg. 4
3: **else** ▷ At least one attribute is categorical
4: $X' \leftarrow \{\ \}$;
5: $Y \leftarrow \{\ \}$;
6: **for** each $z_p \in Z$ **do**
7: $x'_p \leftarrow < id(z_p),\ z_{p,1},\ z_{p,2}, \ldots\ z_{p,d'} >$; ▷ Numerical values
8: $y_p \leftarrow < id(z_p),\ z_{p,d'+1},\ z_{p,d'+2}, \ldots\ z_{p,d} >$; ▷ Categorical values
9: $X' \leftarrow X' \cup \{x'_p\}$;
10: $Y \leftarrow Y \cup \{y_p\}$;
11: **end for**
12: $X'' \leftarrow$ `Metric_to_vector`(Y); ▷ Convert attributes using Alg. 5
13: $X \leftarrow X' \bowtie_{id} X''$;
14: $O \leftarrow$ `HySortOD`$_0(X,\ b,\ MinSplit)$; ▷ Detect outliers using Alg. 4
15: **end if**
16: **return** O;

4.3.1 Time complexity

HySortOD adds an optional, data-conversion step to the method of Sect. 3. If the dataset Z has one or more categorical attributes, then it takes $O(dm)$ time to: (a) isolate the numerical values into a dataset X' and the categorical ones into a dataset Y; (b) extract a nearly equivalent, vector dataset X'' from Y, and; (c) generate a new dataset X by running a join between X' and X''. Particularly for (b), note that we consider that $|\ V_1\ |$ is small. We also consider that the join in (c) is efficiently performed by leveraging the original order in which the instances of Z are stored. If we maintain the same order of instances when storing X', Y and X'', we can then join X' and X'' in $O(dm)$ time by simply matching the instances by their order; no actual matching of identifiers is required. Once the data conversion is complete (or, if it is unnecessary), the basic method HySortOD$_0$ is executed in $O(sn)$ time, in which n is the number of hypercubes generated from X, such that $n \leq m$, and $s = \texttt{Min}(\ \{3^d, n\}\ )$. Consequently, just like the basic method HySortOD$_0$, the overall complexity of HySortOD is $O(sn)$ in the worst case scenario. Note that this analysis relies on the worst-case assumption that all instances are uniformly distributed in space, which rarely occurs in practice because real data tends to be skewed (Faloutsos and Kamel 1994). This phenomenon is further confirmed in our experiments, where the runtime results have a much lower upper bound.

5 Experimental setup

This section presents our experimental setup. HySortOD was coded in Java. It was compared with 9 state-of-the-art competitors, i.e., iForest, HDoutliers, kNN-Out, DB-Out, LOF, ODIN, HilOut, ABOD, and aLOCI. We implemented iForest and HDoutliers in Java for a fair evaluation, Carefully replicating and optimizing the original authors' codes provided in R. The other competitors are also coded in Java, under the framework ELKI (Achtert et al. 2011). For the related works' algorithms that can take advantage of an indexing data structure, we used the default in-memory hashtable expected to be faster than other alternatives[7]. We also used the Euclidean distance for all applicable algorithms.

An exhaustive hyperparameter search was performed to find the configuration that maximizes the effectiveness of each algorithm in each dataset. We also found each algorithm's best-fixed configuration over all datasets. Appendix Appendix A has all the hyperparameter values tested. Importantly, we report the results of two versions of each algorithm. The first uses the algorithm's best-fixed configuration identified. The second uses the hyperparameter values that maximize the resulting effectiveness per dataset. The suffixes "(Best-fixed)" and "(Best)" are added to the algorithms' names to distinguish these cases whenever needed.

[7] https://www.elki-project.github.io/releases/current/javadoc/elki/database/StatiCarrayDatabase.html/

Table 1 Summary of the datasets

Dataset	Instances	Numerical Attributes	Categorical Attributes	Outliers	Outliers (%)
Parkinson	50	22	–	2	4.00
Glass	214	9	–	9	4.21
Ionosphere	237	33	–	12	5.00
Breastw	467	9	–	23	5.00
Pima	526	8	–	26	4.94
Thyroid	3,772	6	–	93	2.47
Satimage2	5,803	36	–	71	1.22
Mammography	11,183	6	–	260	2.32
Shuttle	47,984	9	–	2,399	5.00
Http	567,498	3	–	2,211	0.39
Hepatitis	70	6	14	3	4.29
Tae	107	2	3	5	5.00
Lymphography	148	3	15	6	4.05
Heart	158	7	6	8	5.00
Ecoli	336	5	2	9	2.68
Crx	375	6	9	19	5.00
Australian	403	6	8	20	5.00
Anneal	722	6	4	36	5.00
German	736	7	13	37	5.00
Cmc	1,200	5	4	60	5.00
Car	1,671	-	6	77	5.00
Nursery	12,960	-	8	330	2.54
Adult	23,846	6	8	1,192	5.00
Poker hand	1,025,010	-	10	29,383	2.86

We studied 24 real-world benchmark datasets available online[8,9,10]. They are summarized in Table 1. The datasets with only numerical attributes are shown at the top of the table; those with categorical attributes are presented at the bottom. Many of these datasets are often used by the outlier detection community to evaluate the effectiveness and efficiency of the algorithms. It is valid for all the datasets with only numerical attributes and the datasets `hepatitis` and `ecoli`. The remaining datasets are often used to evaluate classification algorithms. We pre-processed each of them by under-sampling small classes at random, labeling their instances as outliers, and then labeling the instances of other classes as inliers, which is commonly performed in the outlier detection literature (Campos et al. 2016). Specifically, 10

[8] http://odds.cs.stonybrook.edu
[9] https://archive.ics.uci.edu/ml/
[10] https://www.dbs.ifi.lmu.de/research/outlier-evaluation/DAMI/

versions of each dataset were created to account for the randomness of this procedure; we consider the average results.

For the effectiveness evaluation, we used the well-accepted AUROC metric. It ranges between 0 and 1, where a perfect result is scored 1 and a random result is around 0.5. The non-deterministic algorithms were executed 10 times per dataset; we consider the average results. For the efficiency evaluation, we conducted 10 independent executions per algorithm and dataset, recording the execution times and reporting average and standard deviation results. HySortOD, iForest, and HDoutliers are the only algorithms that can process categorical data; they were evaluated considering all the 24 datasets studied. Distinctly, the other algorithms were evaluated on the 10 datasets with only numerical attributes because they cannot handle all the datasets of our study.

All experiments were executed on one single core of a machine with a processor Intel© Xeon™ CPU $E5-2640\ v3\ @\ 2.6GHz$, 2 cores and $16GB$ of RAM, using the Ubuntu 18.04.2 OS. We implemented a time-out limit of 10 hours by aborting the execution of every algorithm requiring more time to finish processing a dataset with a particular configuration. Executions exceeding the main-memory capacity were also aborted. We believe none of the datasets studied is large enough to justify a need for a larger time or space requirement.

Reproducibility as we mentioned before, all codes, detailed results, parameter values tested and datasets used in this paper are freely available for download online, at https://github.com/BraulioSanchez/HySortOD.

6 Results and discussion

This section reports and discusses the results of our experimental evaluation under the setup of Sect. 5. We aimed to answer the following questions:

Q1. Effectiveness and Efficiency – Compared with 9 state-of-the-art algorithms, how effective and efficient is our proposed HySortOD?
Q2. Scalability – How scalable are the evaluated algorithms?
Q3. Parametrization – What are the effects of varying our hyperparameters b and *MinSplit*? And, how to identify an appropriate configuration for using HySortOD in practice?
Q4. Categorical Data – What is the impact of ignoring the categorical attributes of a dataset by processing only the numerical ones?

6.1 Effectiveness and efficiency

This section investigates Question **Q1** by focusing on the effectiveness and efficiency of the studied algorithms. We first evaluate the algorithms in their best configuration, using the hyperparameter values that maximize the effectiveness of every algorithm in each dataset. Later, we consider each algorithm's best-fixed configuration

Table 2 AUROC scores for algorithms that can process categorical data. Best results are shown in bold. HySortOD is more effective than every competitor

Dataset	HySortOD (Best-fixed)	HySortOD (Best)	iForest (Best)	HDoutliers* (Best)
parkinson	0.91	0.98	**1.00**	0.98
glass	0.73	**0.79**	0.68	0.50
ionosphere	0.94	**0.95**	0.93	0.83
breastw	0.97	0.97	**0.99**	0.90
pima	0.73	**0.74**	**0.74**	0.49
thyroid	0.84	0.97	**0.98**	0.95
satimage2	0.98	**0.99**	**0.99**	0.50
mammography	0.74	0.84	**0.87**	0.50
shuttle	0.97	0.97	**0.99**	0.55
http	**0.99**	**0.99**	**0.99**	0.98
hepatitis	0.74	0.77	0.70	**0.83**
tae	0.40	0.62	**0.66**	0.51
lymphography	0.95	**0.98**	0.66	0.81
heart	**0.77**	**0.77**	0.75	0.51
ecoli	0.89	**0.92**	0.33	0.60
crx	0.70	**0.75**	0.66	0.53
australian	0.67	**0.77**	0.69	0.51
anneal	0.81	**0.84**	0.77	0.70
german	0.53	0.54	**0.62**	0.50
cmc	0.40	**0.50**	0.47	0.43
car	0.60	0.65	**0.68**	0.49
nursery	0.60	0.65	**0.83**	0.60
adult	**0.67**	**0.67**	**0.67**	0.50
poker hand	0.50	**0.52**	0.49	⏲
Average rank	2.50 ±0.86	**1.41 ±0.49**	1.95 ±1.13	3.52 ±0.77

*The average rank value only considers reported measurements.

⏲ Execution exceeded the time-out limit.

across all datasets. We also compare our proposed method with the competitors that can process categorical attributes separately from competitors restricted to processing numerical attributes.

Table 2 reports the best AUROC (Area Under the ROC Curve) values obtained from algorithms that can process categorical data.[11] As shown, HySortOD (Best) is more effective than both iForest (Best) and HDoutliers (Best). Its average rank of AUROC values is 1.41, while the corresponding results for iForest (Best) and HDoutliers (Best) are much worse, respectively 1.95 and 3.52. Our proposed algorithm also obtained the largest AUROC values in 14 out of the 24 datasets studied. Note that HDoutliers could not process the dataset `Poker hand` using any hyperparameter configuration due to the time-out limit specified. Table 3 reports the corresponding runtime results. Once again, HySortOD (Best) largely outperformed its competitors, obtaining a *perfect* average ranking of 1.00 with regard to runtime. It was orders of magnitude faster than the others in many cases, especially on large datasets such as `http`, `shuttle`, and `Poker hand`.

[11] For now, we ignore the second columns of Tables 2 and 4. They report the results of our proposal when using the best-fixed configuration across all datasets, which are discussed later.

Table 3 Runtime in seconds for algorithms that can process categorical data. Best results are in bold. Our proposed HySortOD excels in effeciency

Dataset	HySortOD (Best)	iForest (Best)	HDoutliers* (Best)
parkinson	**0.01 ±0.01**	**0.01 ±0.01**	**0.01 ±0.01**
glass	**0.01 ±0.01**	0.03 ±0.01	**0.01 ±0.01**
ionosphere	**0.01 ±0.01**	0.04 ±0.01	**0.01 ±0.01**
breastw	**0.01 ±0.01**	0.05 ±0.01	**0.01 ±0.01**
pima	**0.01 ±0.01**	**0.01 ±0.01**	**0.01 ±0.01**
thyroid	**0.01 ±0.01**	0.24 ±0.01	0.07 ±0.01
satimage2	**0.74 ±0.01**	1.14 ±0.01	1.49 ±0.06
mammography	**0.02 ±0.01**	0.52 ±0.02	0.24 ±0.01
shuttle	**0.01 ±0.01**	3.02 ±0.27	30.37 ±1.20
http	**0.06 ±0.01**	7.77 ±2.05	$3x10^3$ ±27.78
hepatitis	**0.01 ±0.01**	**0.01 ±0.01**	**0.01 ±0.01**
tae	**0.01 ±0.01**	**0.01 ±0.01**	**0.01 ±0.01**
lymphography	**0.01 ±0.01**	**0.01 ±0.01**	**0.01 ±0.01**
heart	**0.01 ±0.01**	**0.01 ±0.01**	**0.01 ±0.01**
ecoli	**0.01 ±0.01**	**0.01 ±0.01**	**0.01 ±0.01**
crx	**0.01 ±0.01**	**0.01 ±0.01**	0.05 ±0.01
australian	**0.01 ±0.01**	0.09 ±0.01	0.07 ±0.01
anneal	**0.01 ±0.01**	**0.01 ±0.01**	0.05 ±0.01
german	**0.01 ±0.01**	0.09 ±0.01	0.20 ±0.01
cmc	**0.01 ±0.01**	0.21 ±0.01	0.14 ±0.01
car	**0.01 ±0.01**	0.02 ±0.01	1.47 ±0.35
nursery	**0.30 ±0.09**	0.86 ±0.27	29.00 ±0.09
adult	**0.13 ±0.01**	0.25 ±0.02	107.37 ±1.95
poker hand	**6.13 ±0.56**	52.87 ±3.84	⏲
Average rank	**1.00 ±0.00**	1.91 ±0.81	1.95 ±0.90

*The average rank value only considers reported measurements.

⏲ Execution exceeded the time-out limit.

We also evaluated our proposal against the competitors that cannot process categorical attributes. Table 4 reports the best AUROC values obtained from the algorithms in each of the 10 datasets having only numerical attributes[11]. HySortOD (Best) is again very effective, this time obtaining the second-best average rank of AUROC values. Its average rank is 2.40, close to the best result of 2.10 obtained by *k*-NN-Out (Best). Our algorithm attained the largest AUROC values in 4 out of the 10 datasets studied. Note that HilOut and aLOCI could not process the dataset `http` using any hyperparameter configuration due to the main-memory limit of our machine. ABOD failed to process the datasets `shuttle` and `http` as it exceeded the specified runtime limit. Table 5 reports the corresponding runtime results. Once more, HySortOD (Best) largely outperformed its competitors. Our algorithm obtained a *quasi-perfect* average ranking of 1.10 concerning the runtime, while the second-best result is 2.60. It was orders of magnitude faster than the others in many cases, especially on the largest datasets `mammography`, `shuttle`, and `http`..

Table 4 AUROC scores for datasets with only numerical attributes. Best results are shown in bold. Our HySortOD is the runner-up in effectivenses

Dataset	HySortOD (Best-fixed)	HySortOD (Best)	kNN-Out (Best)	DB-Out (Best)	LOF (Best)	ODIN (Best)	HilOut* (Best)	aLOCI* (Best)	ABOD* (Best)
parkinson	0.91	**0.98**	0.94	0.95	0.90	0.75	0.96	0.50	0.84
glass	0.73	0.79	0.78	0.78	0.72	0.67	**0.88**	0.63	0.75
ionosphere	0.94	0.95	**0.97**	0.94	0.94	0.93	0.85	0.49	0.85
breastw	0.97	0.97	**0.99**	**0.99**	0.89	0.86	0.88	0.83	**0.99**
pima	0.73	0.74	0.69	0.70	0.67	0.66	**0.78**	0.62	0.66
thyroid	0.84	**0.97**	**0.97**	0.96	0.90	0.85	**0.97**	0.91	0.95
satimage2	0.98	**0.99**	**0.99**	**0.99**	0.98	**0.99**	**0.99**	0.42	**0.99**
mammography	0.74	0.84	0.86	0.87	0.82	0.85	0.84	0.75	**0.88**
shuttle	0.97	0.97	**0.99**	0.69	**0.99**	**0.99**	**0.99**	0.96	⏲
http	**0.99**	**0.99**	**0.99**	**0.99**	**0.99**	0.98	▭	▭	⏲
Average rank	5.20 ±2.48	2.40 ±1.56	**2.10 ±1.44**	3.00 ±2.00	5.10 ±2.34	5.70 ±2.64	2.88 ±2.51	8.33 ±1.05	4.25 ±2.63

*The average rank value only considers reported measurements.

⏲ Execution exceeded the time-out limit.

▭ Execution exceeded the main-memory limit.

Table 5 Runtime in seconds for datasets with only numerical data. Best results are shown in bold. HySortOD is more effecient than every competitor.

Dataset	HySortOD (Best)	kNN-Out (Best)	DB-Out (Best)	LOF (Best)	ODIN (Best)	HilOut* (Best)	aLOCI* (Best)	ABOD* (Best)
parkinson	**0.01±0.01**	**0.01±0.01**	0.02±0.01	0.02±0.01	**0.01±0.01**	0.05±0.01	**0.01±0.01**	0.03±0.01
glass	**0.01±0.01**	0.02±0.01	0.04±0.01	0.03±0.01	0.03±0.01	0.07±0.01	0.04±0.01	0.23±0.01
ionosphere	**0.01±0.01**	0.02±0.01	0.07±0.01	0.05±0.01	0.04±0.01	0.12±0.01	0.04±0.01	0.52±0.01
breastw	**0.01±0.01**	0.03±0.01	0.13±0.01	0.05±0.01	0.05±0.01	0.16±0.01	0.17±0.04	2.75±0.01
pima	**0.01±0.01**	0.05±0.01	0.13±0.01	0.14±0.01	0.08±0.01	0.25±0.01	0.03±0.01	1.33±0.01
thyroid	**0.01±0.01**	0.19±0.01	0.63±0.03	0.59±0.04	0.46±0.03	0.45±0.07	0.31±0.03	884.18±16.33
satimage2	0.74±0.01	1.42±0.07	3.15±0.20	1.78±0.03	3.29±0.03	1.43±0.25	**0.58±0.03**	$3x10^3$±70.40
mammog.	**0.02±0.01**	5.32±0.14	8.89±0.25	2.75±0.32	7.57±0.10	0.62±0.07	6.84±0.18	$2x10^4$±652.06
shuttle	**0.01±0.01**	83.72±1.60	269.98±11.41	135.42±5.32	217.54±7.22	1.66±0.08	8.15±0.16	⏲
http	**0.06±0.01**	$6x10^3$±79.75	$2x10^4$±285.06	$2x10^3$±49.13	$7x10^3$±86.23	▭	▭	⏲
Avg. rank	**1.10±0.30**	2.60±0.91	5.80±0.87	4.30±1.34	4.20±1.72	5.22±2.14	3.33±1.88	7.87±0.33

*The average rank value only considers reported measurements.

⏲ Execution exceeded the time-out limit.

▭ Execution exceeded the main-memory limit.

Now, let us focus again on Fig. 1 from the introductory section. It shows the big picture regarding both efficiency and effectiveness, hence precisely answering Question **Q1**. We summarize in this figure the results obtained for all 24 datasets, reporting average runtime ranking versus average AUROC ranking for HySortOD (black circles) and the 9 competitors in their best configuration. Results for all the datasets studied and for those datasets with only numerical attributes are shown separately on the left side (a) and on the right side (b) of the figure, respectively. An orange cross at the origin of the plots marks the position of a fictitious ideal detector that would rank first in both runtime and AUROC. Note that our HySortOD consistently outperformed all of the 9 state-of-the-art competitors, being clearly the one that is the closest to the ideal detector. Additionally, note that LOF, aLOCI, *k*NN-Out, ODIN, ABOD, DB-Out, and HilOut failed, i.e., could not even be tested, in the datasets with categorical attributes. These results reveal a significant advantage of our HySortOD over the other detectors.

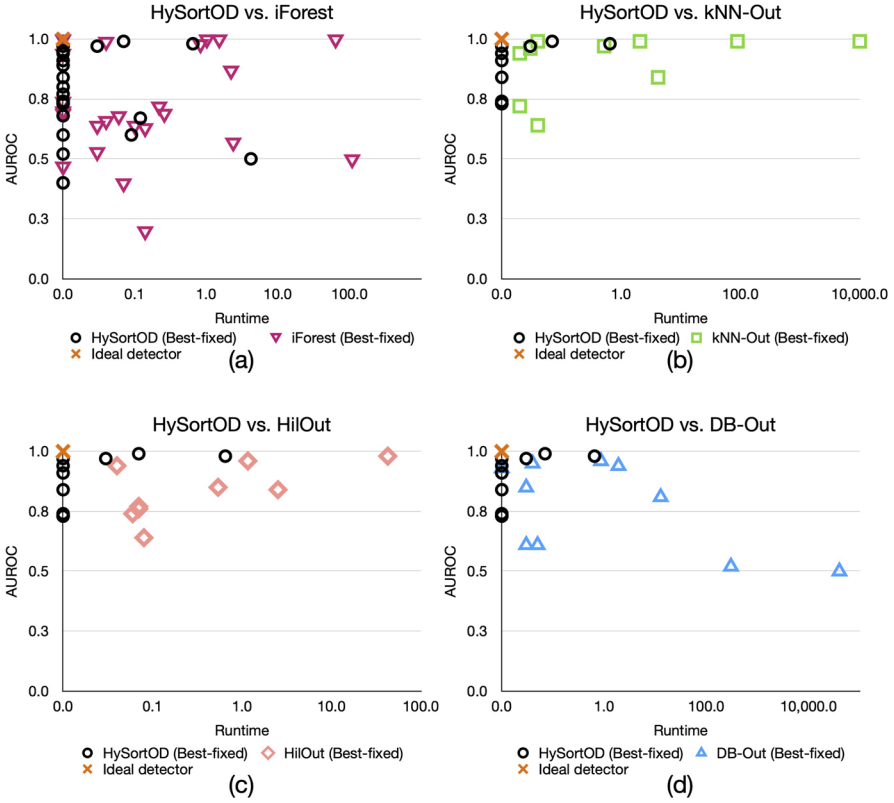


Fig. 4 Runtime and AUROC values obtained per dataset using the 'Best-fixed' versions of algorithms HySortOD, iForest, *k*NN-Out, HilOut, and DB-Out. A fictitious ideal detector (orange crosses) would always obtain a maximal AUROC score in zero time. As shown, our algorithm (black circles) is clearly the one that is the closest to the ideal detector for most datasets

At last, we evaluate the algorithms using their best-fixed configuration of hyper-parameters across all datasets. We consider only the most competitive algorithms in this analysis for brevity. As shown in Tables 2 and 4, our algorithm attained a better average AUROC ranking than HDoutliers (Best), ODIN (Best), and aLOCI (Best) even when using its best-fixed configuration. Evaluating these competitors' 'Best-fixed' versions is, therefore, unnecessary. We also consider evaluating the 'Best-fixed' versions of ABOD and LOF to be unnecessary. The former has a cubic time complexity concerning the data Cardinality, and the latter is also inefficient compared with our algorithm. LOF also obtained an average AUROC ranking of 5.10 using its best configuration per dataset. It is then unlikely that LOF (Best-fixed) will outperform our HySortOD (Best-fixed)'s result of 5.20. Hence, we evaluate the 'Best-fixed' versions of algorithms HySortOD, iForest, *k*NN-Out, HilOut, and DB-Out.

Figure 4 reports the runtime versus the AUROC value obtained per dataset from these five algorithms. We plot runtime and AUROC values instead of average

ranking values to provide the most detailed result possible. An orange cross (this time, at the top-left of each plot, not at the origin) marks the position of a fictitious ideal detector that would obtain a maximal AUROC score in zero time for any dataset. Note that HySortOD (Best-fixed) generally outperforms every competitor by being the one that is the closest to the ideal detector for most datasets. We therefore conclude that our algorithm balances effectiveness and efficiency better than its competitors both when considering the best configuration per dataset and the best-fixed configuration across datasets.

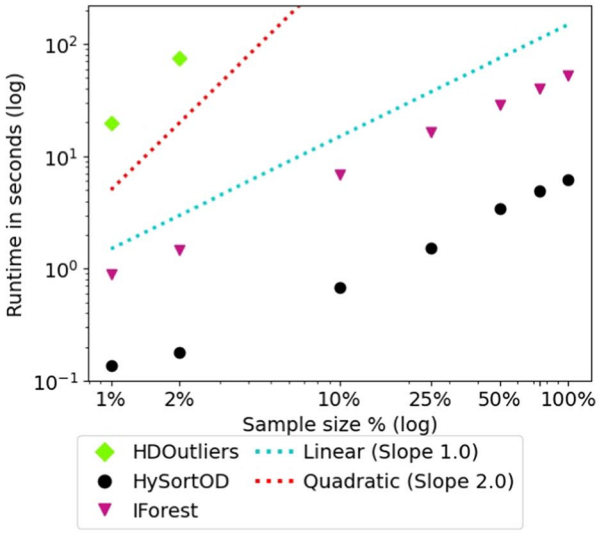
6.2 Scalability

This section investigates Question **Q2** by focusing on the scalability of the algorithms studied. One of the main challenges in outlier detection is the capacity to process large data in a feasible time. To our knowledge, no labeled dataset for outlier detection is larger in Cardinality than `Poker hand` and `http`. Hence, we demonstrate the scalability of our algorithm by further investigating these datasets. The former contains only categorical attributes; the latter has only numerical ones. To this end, we created smaller versions of `Poker hand` and `http` with random samples of increasing sizes up to the entire datasets, i.e., 1%, 2%, 10%, 25%, 50%, 75% and 100%, and report the corresponding average runtime of each algorithm. Note that we randomly sampled proportional percentages of inlier and outlier instances to keep the original data characteristics. Every algorithm was tested with the hyperparameter configuration that maximizes its effectiveness in each dataset's full version, i.e., the 100% version, except for HDoutliers in `Poker hand` and aLOCI in `http` because they failed in these datasets due to the time-out or the main-memory limits. In these cases, we used the default setting recommended in the original publication. aLOCI was configured with $\alpha = 1/16$, $g = 20$, and $nmin = 20$; HDoutliers used $\alpha = 0.05$.

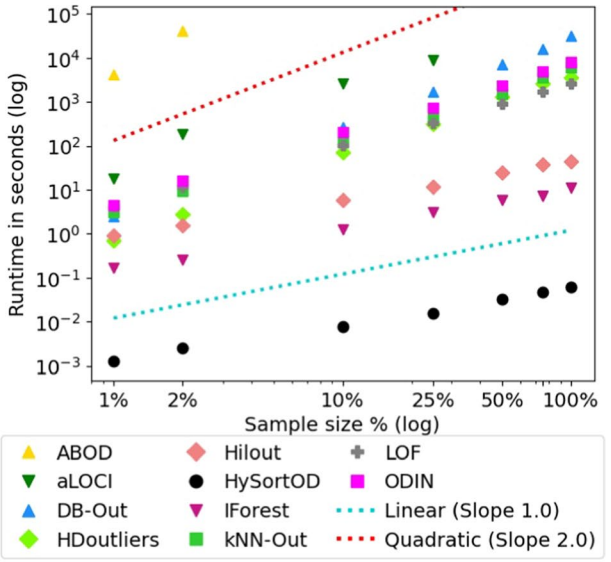
Figure 5 reports the corresponding results. Note that HDoutliers exceeded the time-out limit regarding `Poker hand` when the sample size was larger than 2%. ABOD executed successfully in `http` with up to 2% sample size but exceeded the time-out limit with larger sizes. aLOCI exceeded the main-memory capacity in `http` with sample sizes larger than 25%. The other related work algorithms required massive amounts of time to complete – notice that the plots are in *log-log* scale. In contrast, HySortOD obtained much better results. Overall, our algorithm is scalable and outperformed *every* competitor by a large margin in both datasets. These findings suggest that the use of our new hypercube-ordering-and-searching strategy to detect outliers is indeed a powerful scale-up tool.

6.3 Parametrization

This section investigates Question **Q3** by focusing on the parametrization of HySortOD. Our algorithm has two hyperparameters, b and *MinSplit*. The general intuition behind b is that, as it increases, the hypercube length $l = \frac{1}{b}$



(a) Scalability on categorical data from `poker hand`.



(b) Scalability on numerical data from `http`.

Fig. 5 Runtime on random samples of datasets (a) `Poker hand` and (b) `http`. We use the hyperparameter configurations that maximize the effectiveness of every algorithm in each dataset's full version, i.e., 100% version, except for HDoutliers in `Poker hand` and aLOCI in `http` because they fail in these datasets and thus use a default configuration. Overall, HySortOD is scalable and outperforms *every* competitor by a large margin in both (a) and (b). Notice that the plots are in *log-log* scale

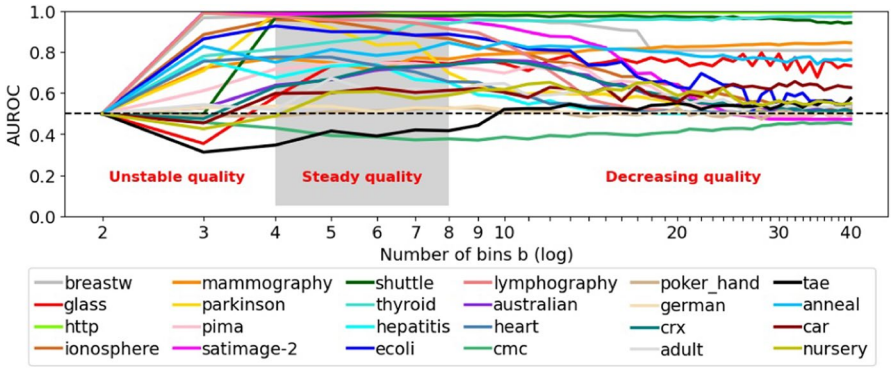


Fig. 6 AUROC versus number of bins b for 24 real datasets. There are three distinct intervals with different patterns: "Unstable", "Steady" and "Decreasing quality". The best results are likely to be in the "Steady quality" interval, so we suggest using $b = 5$ as the default configuration of HySortOD

decreases, and so does the radius of the neighborhood analyzed when searching for the outliers. Consequently, b influences the effectiveness of HySortOD. For large values of b, more instances are expected to be reported as outliers because they may lie alone in a small neighborhood. By contrast, for small values of b, it is expected less instances are reported as outliers because the hypercubes are likely large enough to make nearly every instance a neighbor of many other instances. We studied the effect of b by varying its value from 2 to 40 for all the 24 datasets. Figure 6 reports our findings; it plots AUROC versus b. Note that the horizontal axis is shown in log scale to improve the visualization. Overall, there are three distinct intervals with different patterns in this axis. We name the intervals according to their general trend, that is: "Unstable", "Steady", and "Decreasing quality" in the intervals [2, 3], [4, 8], and [9, 40], respectively. Based on these observations, we understand the "Steady quality" interval as the appropriate one to obtain high-quality results in a general scenario, which is corroborated by 24 real datasets of distinct domains. Hence, we suggest $b = 5$ as the default configuration of HySortOD.

We also analyzed different variations for our hyperparameter *MinSplit*. It only affects runtime, *not* accuracy. *MinSplit* balances the number of hypercubes mapped by each node of the dimension-wise tree, thus limiting the number of hypercubes to scan during the neighborhood searches. In general, small values create leaf nodes that map only a few hypercubes; large values have the opposite effect, which tends to degenerate to the sequential scan runtime. With that in mind, we executed experiments varying the hyperparameter from 10 to 1, 000 in each of the 24 datasets and observe that setting the value 100 generally reduces the runtime compared to using other values. We therefore suggest setting $MinSplit = 100$ as the default configuration of HySortOD.

6.4 Categorical data

This section investigates Question **Q4**. We aim to understand the impact of ignoring the categorical attributes of a dataset by processing only the numerical ones. To this end, we first study our real-world benchmark datasets considering practical applications. Then, we analyze synthetic data to better investigate the use of categorical attributes in a controlled environment. All experiments of this section employ our proposed algorithm HySortOD configured with the default hyperparameter values $b = 5$ and $MinSplit = 100$.

Table 6 provides results of effectiveness for every real dataset having at least one categorical attribute. For each dataset, we report the AUROC score obtained when processing only numerical attributes, and the score obtained from both numerical and categorical attributes; see columns "Numerical" and "Numerical & Categorical", respectively. We also report the difference between these two scores in column "Difference". The datasets `Car`, `Nursery` and `Poker hand` have no numerical attribute; Hence, the "Numerical" column shows the expected AUROC score of a random ranking for these datasets. Gains and losses in the effectiveness are respectively depicted in green and red. As expected, the categorical data often provided gains and rarely caused losses. Effectiveness gains occured in 9 out of the 14 datasets studied and they were expressive in some cases, as for `ecoli`, `lymphography`, `Car`, and `Nursery` where the scores improved up to 0.27. Distinctly, the losses were minor and rare; they occurred for `hepatitis`, `heart` and `anneal` where the scores reduced no more than 0.06. Thus, processing the categorical data was generally advantageous.

Table 6 AUROC scores with and without considering categorical attributes

Dataset	Numerical N	Numerical & Categorical NC	Difference $NC - N$
hepatitis	**0.80**	0.74	-0.06
tae	0.36	**0.40**	0.05
lymphography	0.75	**0.95**	0.20
heart	**0.78**	0.77	-0.01
ecoli	0.62	**0.89**	0.27
crx	0.64	**0.70**	0.06
australian	0.60	**0.67**	0.07
anneal	**0.84**	0.81	-0.03
german	**0.53**	**0.53**	0.00
cmc	0.37	**0.40**	0.03
car	0.50*	**0.60**	0.10
nursery	0.50*	**0.60**	0.10
adult	0.66	**0.67**	0.01
poker hand	**0.50***	**0.50**	0.00

* Expected AUROC score of a random ranking

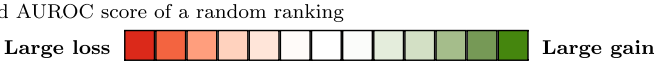


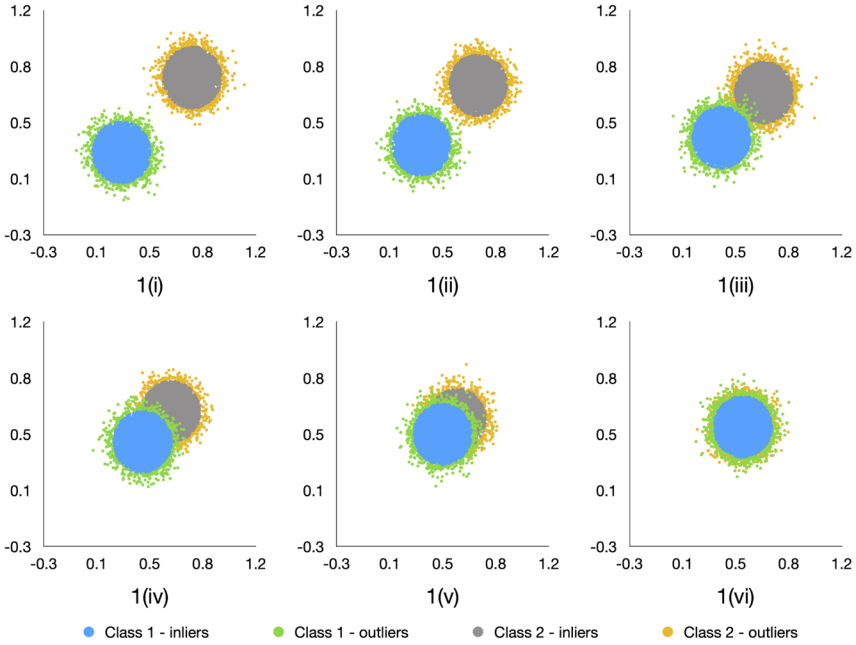
Fig. 7 Scenarios used to study the impact of categorical data. (a): Scenario 1 has 2 classes getting closer to one another in the numerical data. Note that some outliers in green/yellow are undetectable in the Datasets `1 (iii)`, `1 (iv)` and `1 (v)`. (b): Scenario 2 has 2 classes with complete overlap in the numerical data. Except for Dataset `2 (vi)`, every outlier in green is undetectable

Note that any attribute of a dataset may contain relevant or irrelevant information regarding the detection of outliers, and it applies to both numerical and categorical attributes. A relevant attribute should have values for the inliers that are considerably different from those of the outliers. On the other hand, an attribute with a randomly selected value for each instance regardless of its type is probably irrelevant. Importantly, irrelevant attributes may even hurt the effectiveness of the detection because they dilute the useful information of other attributes with useless information. We believe it is the case of the datasets `hepatitis`, `heart` and `anneal` whose AUROC scores are slightly larger when ignoring their (probably irrelevant) categorical attributes than the scores obtained with these attributes. Hence, we leveraged synthetic data to better understand the use of categorical attributes in a controlled environment.

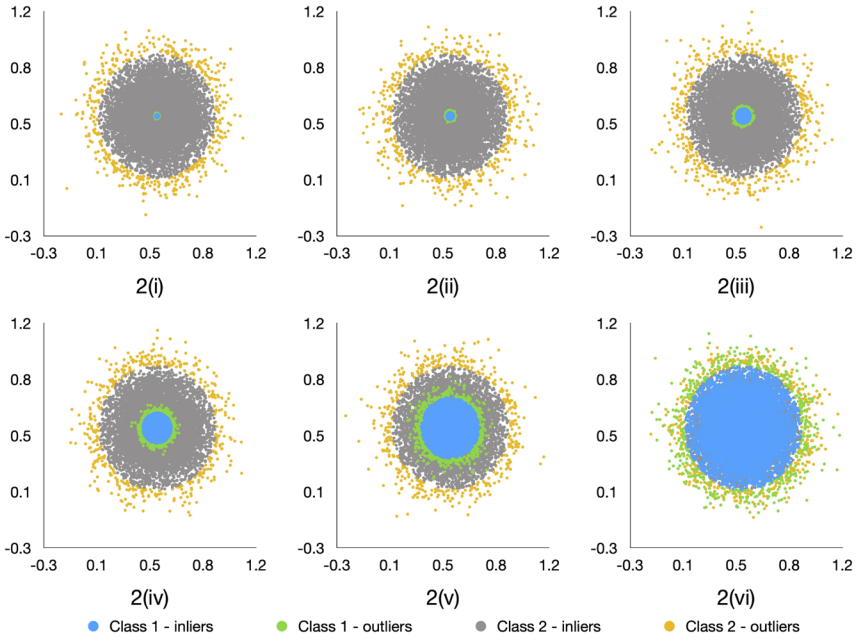
We intended to generate synthetic datasets having similar metadata as the real datasets of the previous experiment. On average, the real datasets have 18, 758 instances described by 6 numerical attributes and 9 categorical attributes, with 4.6% of outliers. The average number of classes is 2, which is known because the datasets were originally created for classification. These average values were then used to guide the generation of the synthetic data.

We considered two synthetic scenarios with 6 numerical attributes that are not always informative enough to detect every outlier, and then evaluated the impact of adding 9 relevant categorical attributes to the data. Later, we also evaluated the impact of adding 9 irrelevant categorical attributes on top of the relevant ones. Scenario 1 has 2 classes getting closer to one another with increasing overlap in the numerical data according to six datasets, namely Datasets `1(i)`, `1(ii)`, `1(iii)`, `1 (iv)`, `1 (v)` and `1 (vi)`. Each dataset has 18, 758 instances out of which 4.6% are outliers. Both classes follow Gaussian distributions in every numerical attribute; their mean values are considerably distinct in the Dataset `1 (i)` and get more and more similar in the other datasets, until they become equal in the Dataset `1 (vi)`. Figure 7(a) plots two of the numerical attributes of each dataset. Note that some green and yellow outliers are close to many inliers of the other class in Datasets `1 (iii)`, `1 (iv)` and `1 (v)`. They are, therefore, undetectable in the numerical data. Distinctly, each relevant categorical attribute allows distinguishing the outliers by having values employed exclusively for them. The irrelevant categorical attributes contain values selected at random, though.

We also created modified versions of Dataset `1 (i)` where we make the problem harder by replacing some of the numerical attributes with other numerical attributes that contain random values. Specifically, six modified versions of this dataset where created with the number of numerical attributes replaced varying from 0 to 5. No other modification was performed to the data. Figure 8 plots two of the numerical



(a) Scenario 1: 2 classes getting closer to one another



(b) Scenario 2: 2 classes with complete overlap

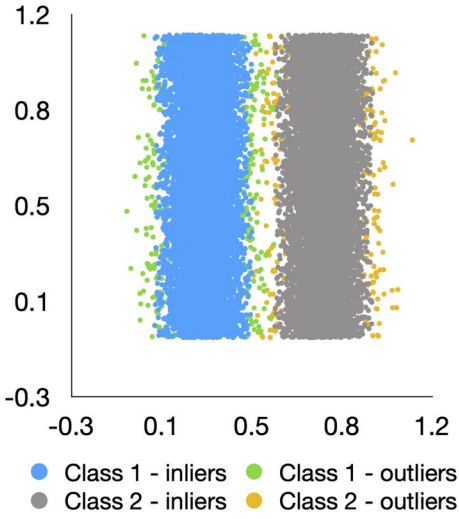
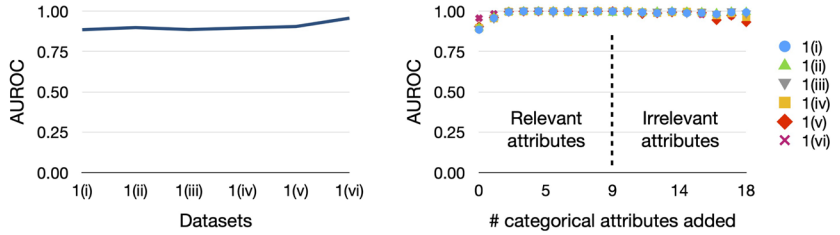


Fig. 8 Scenario 1(i) + irrelevant: Dataset `1 (i)` adapted to have one numerical attribute that is relevant and another attribute that is irrelevant

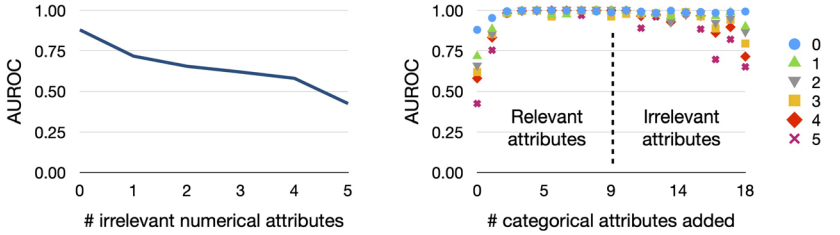
attributes of one of these modified datasets. Note that the vertical attribute has random values and is, therefore, irrelevant to detect the outliers. Distinctly, the horizontal attribute contains information relevant to the detection. We refer to this particular variation of the first scenario as "Scenario 1(i) + irrelevant". The same procedure could also be applied to the other datasets of this scenario and even to those of the second scenario that is presented later, but we only consider Dataset `1 (i)` of Scenario 1 for brevity.

Now, we move on to our second scenario. It regards a more extreme situation where a large portion of the outliers cannot be detected in the numerical data because of a complete overlap of classes. Similar to the previous scenario, our Scenario 2 presents 2 classes that follow Gaussian distributions in each numerical attribute. The difference is that their mean values are now the same and the standard deviation of one class increases according to Datasets `2 (i)` to `2 (vi)` until it matches the standard deviation of the other class. Once more, every dataset has 18, 758 instances described by 6 numerical attributes that may not be enough to detect all the 4.6% of outliers; And, we evaluate the impact of adding 9 relevant categorical attributes at first, and 9 irrelevant categorical attributes on top of the relevant ones later. Figure 7(b) plots two numerical attributes of each dataset. Note that every green outlier in Datasets `2 (i)` to `2 (v)` is close to inliers of the other class. Hence, they cannot be detected in the numerical data. We therefore depend on the categorical data, which were generated using the same procedure of the previous scenario.

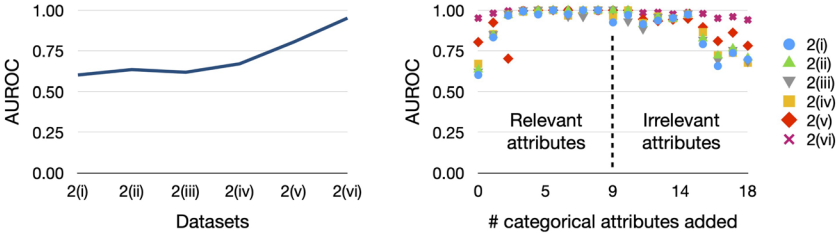
Figure 9 provides results of effectiveness for the datasets of all scenarios. The plots on the left side of the figure show the AUROC scores obtained exclusively from the numerical data. Distinctly, the right-sided plots present the corresponding scores obtained when analyzing categorical attributes together with the numerical ones. Figure 9(a) depicts the Scenario 1. As expected, sub-optimal scores of nearly 0.85 were obtained from the numerical data of most datasets of this scenario, except for Dataset `1 (vi)` where better scores were attained because the task is clearly



(a) Scenario 1: AUROC scores for the datasets of the scenario, considering only numerical attributes (left) and both numerical and categorical attributes (right).



(b) Scenario 1(i) + irrelevant: AUROC for Dataset `1(i)` with some numerical attributes that are irrelevant. Left: numerical only; Right: numerical and categorical.



(c) Scenario 2: AUROC scores for the datasets of the scenario, considering only numerical attributes (left) and both numerical and categorical attributes (right).

Fig. 9 Studying the impact of categorical data. Left: AUROC scores obtained with numerical data only. Right: The corresponding scores when analyzing relevant and irrelevant categorical attributes together with the numerical ones

easier. The use of categorical attributes together with the numerical ones also led to the results we expected. Adding a few relevant categorical attributes to the analysis was already enough to achieve near-perfect scores. And, the irrelevant categorical attributes added on top of the relevant ones indeed deteriorated the effectiveness of the detection. Interestingly, it seems that each irrelevant categorical attribute canceled the effects of a relevant one, and, therefore, the scores roughly degraded to their original values when all the irrelevant attributes were considered.

Figure 9(b) regards "Scenario 1(i) + irrelevant". Again, the results matched our expectations. The larger the number of irrelevant numerical attributes, the worse are the scores obtained when using only the numerical portion of each dataset. A few relevant

categorical attributes were once more enough to improve the scores to a near-perfect status, even when there is only one relevant numerical attribute and the score is bellow 0.5. Once again, each irrelevant categorical attribute added to the analysis nearly canceled the effects of a relevant one.

Finally, the results of the Scenario 2 are shown in the Fig. 9(c). They also match our expectations with increasingly better scores being obtained from the numerical attributes as we vary the dataset analyzed, from Datasets `2 (i)` to `2 (vi)`. The use of categorical data also led to the results we expected. The relevant categorical attributes were again capable of improving the scores to near-perfect scores, and the irrelevant ones nearly canceled relevant attributes.

In summary, this section investigated the ability of HySortOD to capitalize on categorical data. Overall, we confirmed through an extensive experimental evaluation that relevant categorical attributes can greatly improve the effectiveness of our method in both real and synthetic data. We also verified that numerical and categorical attributes containing irrelevant values may deteriorate the results obtained. Irrelevant attributes should, therefore, be avoided regardless of their type. We believe it can be performed by leveraging the knowledge of field experts that can identify and use only the most appropriate attributes to each particular application, which applies to outlier detection and also so any other analytical task in general.

7 Conclusion

Outlier detection has many applications and is, therefore, covered by an extensive literature. The distance-based detectors are the most popular ones. Despite this popularity, these detectors still have two major drawbacks that we identified and demonstrated through a comprehensive experimental evaluation: (a) the intensive neighborhood search that takes hours or even days to complete in large data, and; (b) the inability to process categorical attributes. This paper tackled both problems with the new method HySortOD to detect outliers. Here, we argue that efficiency and generality of data type are very desirable aspects to be considered in any analytical task, including outlier detection.

Our main contributions are:

C1 – **Efficiency**: We Carefully designed HySortOD- a novel algorithm that efficiently identifies outliers using a new hypercube-ordering-and-searching approach that speeds up the detection task to work at scale. Its main focus is the analysis of datasets with many instances and a low-to-moderate number of attributes.
C2 – **Generality**: We introduced a pivot-based strategy that allows our method to analyze both numerical and categorical attributes seamlessly.
C3 – **Benchmark Evaluation**: We studied 24 real-world, benchmark datasets with up to 1 million instances, and showed that our method outperforms 9 competitors of the state of the art in runtime, being up to 6 orders of magnitude faster, while still presenting very accurate results.

C4 – **Case Study**: We investigated the ability of our method to capitalize on categorical attributes, and confirmed through an extensive experimental evaluation that these attributes can greatly improve the accuracy of the detection of outliers in both real and synthetic datasets.

Finally, let us highlight that a hypercube-ordering-and-searching approach similar to ours has been used previously, in the distinct context of similarity retrieval (Xia et al. 2004; Kalashnikov 2013). To our knowledge, we are the first ones to use it for outlier detection. Hence, we consider the fact that we found one new usage for a tool from a distinct area of research to be an additional contribution of our work.

Appendix A

Model configurations

Tables 7 and 8 report the hyperparameter values tested to find the configuration that maximizes the effectiveness of every algorithm in each dataset and the best-fixed configuration per algorithm across all datasets.

Table 7 Hyperparameter values used for algorithms working with both categorical and numerical data

Dataset	HySortOD	iForest	HDoutliers
parkinson	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
glass	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
ionosphere	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
breastw	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
pima	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
thyroid	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
satimage2	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
mammography	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
shuttle	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
http	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
hepatitis	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
tae	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
lymphography	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
heart	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
ecoli	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
crx	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
australian	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
anneal	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
german	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
cmc	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
Car	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
Nursery	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
Adult	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$
Poker hand	$b \in \{2, 3, \ldots, 100\}$	$h \in \{100, 150, \ldots, 300\}$, $\psi \in \{32, 64, \ldots, 1024\}$	$\alpha \in \{0.05, 0.06, \ldots, 0.5\}$

Table 8 Hyperparameter values used for algorithms working with only numerical data.

Dataset	kNN-Out	DB-Out	LOF	ODIN	HilOut	aLOCI
parkinson	$k \in \{1, 2, \ldots, 50\}$	$d \in \{1, 2, \ldots, 50\}$	$k \in \{1, 2, \ldots, 50\}$	$k \in \{1, 2, \ldots, 50\}$	$k \in \{1, 2, \ldots, 10\}$, $h \in \{1, 2, \ldots, 64\}$	$n \in \{1, 2, \ldots, 10\}$, $g \in \{1, 2, 3, 4\}$
glass	$k \in \{1, 2, \ldots, 20\}$	$d \in \{1, 2, \ldots, 50\}$	$k \in \{1, 2, \ldots, 50\}$	$k \in \{1, 2, \ldots, 50\}$	$k \in \{1, 2, \ldots, 100\}$, $h \in \{1, 2, \ldots, 64\}$	$n \in \{1, 2, \ldots, 200\}$, $g \in \{1, 2, 3, 4\}$
ionosphere	$k \in \{1, 2, \ldots, 50\}$	$d \in \{1, 2, \ldots, 50\}$	$k \in \{1, 2, \ldots, 50\}$	$k \in \{1, 2, \ldots, 50\}$	$k \in \{1, 2, \ldots, 20\}$, $h \in \{2, 4, \ldots, 50\}$	$n \in \{1, 2, \ldots, 50\}$, $g \in \{1, 2, 3, 4\}$
breastw	$k \in \{1, 2, \ldots, 50\}$	$d \in \{1, 2, \ldots, 50\}$	$k \in \{1, 2, \ldots, 50\}$	$k \in \{1, 2, \ldots, 50\}$	$k \in \{1, 2, \ldots, 20\}$, $h \in \{2, 3, \ldots, 32\}$	$n \in \{1, 2, \ldots, 10\}$, $g \in \{1, 2, 3, 4\}$
pima	$k \in \{100, 101, \ldots, 500\}$	$d \in \{10, 11, \ldots, 200\}$	$k \in \{100, 101, \ldots, 500\}$	$k \in \{1, 2, \ldots, 300\}$	$k \in \{100, 110, \ldots, 500\}$, $h \in \{1, 2, \ldots, 20\}$	$n \in \{1, 2, \ldots, 100\}$, $g \in \{1, 2, 3, 4\}$
thyroid	$k \in \{1, 2, \ldots, 50\}$	$d \in \{1, 2, \ldots, 50\}$	$k \in \{50, 51, \ldots, 150\}$	$k \in \{1, 2, \ldots, 100\}$	$k \in \{1, 2, \ldots, 20\}$, $h \in \{1, 2, \ldots, 64\}$	$n \in \{1, 2, \ldots, 50\}$, $g \in \{1, 2, 3, 4\}$
satimage2	$k \in \{1, 2, \ldots, 100\}$	$d \in \{10, 20, \ldots, 200\}$	$k \in \{50, 51, \ldots, 150\}$	$k \in \{10, 20, \ldots, 2000\}$	$k \in \{1, 2, \ldots, 50\}$, $h \in \{1, 2, \ldots, 64\}$	$n \in \{1, 2, \ldots, 10\}$, $g \in \{1, 2, 3, 4\}$
mammography	$k \in \{100, 110, \ldots, 2500\}$	$d \in \{1, 2, \ldots, 10\}$	$k \in \{100, 101, \ldots, 250\}$	$k \in \{100, 150, \ldots, 2000\}$	$k \in \{1, 2, \ldots, 50\}$, $h \in \{1, 2, \ldots, 64\}$	$n \in \{1, 2, \ldots, 10\}$, $g \in \{1, 2, 3, 4\}$
shuttle	$k \in \{2000, 2100, \ldots, 5000\}$	$d \in \{1000, 1050, \ldots, 3000\}$	$k \in \{2000, 2100, \ldots, 5000\}$	$k \in \{1000, 1100, \ldots, 12000\}$	$k \in \{1, 2, \ldots, 20\}$, $h \in \{1, 2, \ldots, 64\}$	$n \in \{1, 101, \ldots, 3001\}$, $g \in \{1, 2, 3, 4\}$
http	$k \in \{2000, 2100, \ldots, 4000\}$	$d \in \{1000, 1050, \ldots, 3000\}$	$k \in \{2000, 2100, \ldots, 4000\}$	$k \in \{2000, 2100, \ldots, 7000\}$	$k \in \{10, 20, \ldots, 300\}$, $h \in \{1, 2, \ldots, 64\}$	$n \in \{1, 101, \ldots, 5001\}$, $g \in \{1, 2, 3, 4\}$

Acknowledgements This work was supported in part by the São Paulo Research Foundation (FAPESP) – Grant No 2018/05714-5, 2016/17078-0 and 2020/07200-9, by the Coordination for Improvement of Higher Education Personnel (CAPES) – Grant No 132788/2018-7, and by the National Council for Scientific and Technological Development (CNPq).

Funding Open Access funding provided by Carnegie Mellon University.

Code availability As we mentioned before, all codes, detailed results, parameter values tested and datasets used in this paper are freely available for download online, at https://github.com/BraulioSanchez/HySortOD

References

Jauhri A, McDanel B (2015) Connor Chris (2015) Outlier Detection for Large Scale Manufacturing Processes. In: IEEE Big Data. (pp. 2771–2774)

Anbarasi MS, Dhivya S (2017) Fraud Detection Using Outlier Predictor in Health Insurance data. In: ICICES, p. 6

Tripathi D, Lone T, Sharma Y, Dwivedi S (2018) Credit Card fraud detection using local outlier factor. IJPAM 118(7):229–234

Bindu PV, Thilagam P, Ahuja D (2017) Discovering suspicious behavior in multilayer social networks. Comput Human Behav 73:568–582

Jabez J, Muthukumar B (2015) Intrusion Detection System (ids): Anomaly Detection Using Outlier Detection Approach. In: ICICC, pp. 338–346

Shahid N, Naqvi IH, Qaisar SB (2015) Characteristics and classification of outlier detection techniques for wireless sensor networks in harsh environments: survey. Artif Intell Rev 43:193–228

Campos G, Zimek A, Sander J, Campello R, Micenková B, Schubert E, Assent I, Houle ME (2016) On the evaluation of unsupervised outlier detection: measures, datasets, and an empirical study. DMKD 30(4):891–927

Aggarwal CC (2017) Outlier Analysis. Springer, Switzerland

Grubbs FE (1950) Sample criteria for testing outlying observations. Ann Math Statis 21(1):27–58

Knorr EM, Ng RT (1998) Algorithms for mining distance-based outliers in large datasets. In: Proceedings of the 24rd International Conference on Very Large Data Bases. Morgan Kaufmann Publishers Inc: San Francisco (pp. 392–403)

Ramaswamy S, Rastogi R, Shim K (2000) Efficient algorithms for mining outliers from large data sets. SIGMOD 29(2):427–438

Breunig MM, Kriegel H-P, Ng RT, Sander J (2000) LOF: Identifying density-based local outliers. SIGMOD 29(2):93–104

Angiulli F, Pizzuti C (2002) Fast Outlier Detection in High Dimensional Spaces. In: European Conference on PKDD, pp. 15–27 (2002)

Papadimitriou S, Kitagawa H, Gibbons PB, Faloutsos C (2003) LOCI Fast Outlier Detection. In: ICDE, pp. 315–326

Hautamäki V, Kärkkäinen I, Fränti P (2004) Outlier Detection Using k-Nearest Neighbour Graph. In: ICPR, pp. 430–433. IEEE

Amagata D, Onizuka M, Hara T. Fast and exact outlier detection in metric spaces: a proximity graph-based approach. InProceedings of the 2021 International Conference on Management of Data 2021 Jun 9 (pp. 36-48)

Amagata D, Onizuka M, Hara T (2022) Fast, exact, and parallel-friendly outlier detection algorithms with proximity graph in metric spaces. VLDB J. 31(4):797–821

Orair GH, Teixeira CHC, Meira W, Wang Y, Parthasarathy S (2010) Distance-based outlier detection: consolidation and renewed bearing. VLDB Endow 3(1–2):1469–1480

Goldstein M, Uchida S (2016) A comparative evaluation of unsupervised anomaly detection algorithms for multivariate data. PLoS ONE 11(4):1–31

Kirner E, Schubert E, Zimek A (2017) Good and Bad Neighborhood Approximations for Outlier Detection Ensembles. In: SISAP, pp. 173–187

Akoglu L, Tong H, Vreeken J, Faloutsos C (2012) Fast and reliable anomaly detection in categorical data. In: Proceedings of the 21st ACM International Conference on Information and Knowledge Management. CIKM '12, pp. 415–424. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2396761.2396816. https://doi.org/10.1145/2396761.2396816

Tang G, Bailey J, Pei J, Dong G (2013) Mining multidimensional contextual outliers from categorical relational data. In: Proceedings of the 25th International Conference on Scientific and Statistical Database Management. SSDBM. Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2484838.2484883. https://doi.org/10.1145/2484838.2484883

Harris DM, Harris SL (2012) Digital Design and Computer Architecture. Morgan Kaufmann, Burlington

Wilkinson L (2018) Visualizing big data outliers through distributed aggregation. IEEE Trans. Vis. Comput. Graph. 24(1):256–266. https://doi.org/10.1109/TVCG.2017.2744685

Liu FT, Ting KM, Zhou Z (2008) Isolation forest. In: Proceedings of the 8th IEEE International Conference on Data Mining (ICDM 2008), December 15-19, 2008, Pisa, Italy, pp. 413–422. IEEE Computer Society. https://doi.org/10.1109/ICDM.2008.17. https://doi.org/10.1109/ICDM.2008.17

Liu FT, Ting KM, Zhou Z (2012) Isolation-based anomaly detection. ACM Trans. Knowl. Discov. Data 6(1):3–1339. https://doi.org/10.1145/2133360.2133363

Schubert E, Zimek A, Kriegel HP (2014) Local outlier detection reconsidered: a generalized view on locality with applications to spatial, video, and network outlier detection. DMKD 28(1):190–237

Fraideinberze AC, Rodrigues JF, Cordeiro RLF (2016) Effective and Unsupervised Fractal-based Feature Selection for Very Large Datasets: removing linear and non-linear attribute correlations. In: ICDM Workshops, pp. 615–622 (2016)

Traina Junior C, Traina AJM, Faloutsos C, Seeger B (2002) Fast indexing and visualization of metric data sets using slim-trees. TKDE 14(2):244–260

Mo D, Huang SH (2012) Fractal-based intrinsic dimension estimation and its application in dimensionality reduction. TKDE 24(1):59–71

Tran L, Fan L, Shahabi C (2016) Distance-based outlier detection in data streams. VLDB Endow. 9(12):1089–1100

Yoon S, Lee J-G, Byung BS (2018) NETS: Extremely fast outlier detection from a data stream via set-based processing. Proceed VLDB Endow. 12:1303–1315

Cabral EF, Cordeiro RLF (2020) Fast and scalable outlier detection with sorted hypercubes. In: Proceedings of the 29th ACM International Conference on Information & Knowledge Management. CIKM '20, pp. 95–104. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3340531.3412033

Faloutsos C, Kamel I (1994) Beyond uniformity and independence: Analysis of r-trees using the concept of fractal dimension. In: Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. PODS '94, pp. 4–13. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/182591.182593. https://doi.org/10.1145/182591.182593

Robinson DJS (2003) An Introduction to Abstract Algebra, 2nd edn. Walter de Gruyter, Berlin

Marcus B (2022) Lecture Notes, Math 342: Algebra and Coding Theory. https://personal.math.ubc.ca/~marcus/Math342/Math342_Lectures3-4.pdf. Day of access: 17 Nov

Traina Junior C, Santos Filho RF, Traina AJM, Vieira MR, Faloutsos C (2007) The omni-family of all-purpose access methods: a simple and effective way to make similarity search more efficient. VLDB J. 16(4):483–505. https://doi.org/10.1007/s00778-005-0178-0

Schroeder MR (1991) Fractals, Chaos, Power Laws. Minutes from an Infinite Paradise. W H Freeman, New York

Rissanen J (1983) A universal prior for integers and estimation by minimum description length. Ann Statis 11(2):416–431. https://doi.org/10.1214/aos/1176346150

Chakrabarti D, Papadimitriou S, Modha DS, Faloutsos C (2004) Fully automatic cross-associations. In: KDD, pp. 79–88. ACM

Achtert E, Hettab A, Kriegel HP, Schubert E, Zimek A. Spatial outlier detection: Data, algorithms, visualizations. InAdvances in Spatial and Temporal Databases: 12th International Symposium, SSTD 2011, Minneapolis, MN, USA, August 24-26, 2011, Proceedings 12 2011 (pp. 512-516). Springer: Berlin

Xia C, Lu H, Ooi BC, Hu J. Gorder: an efficient method for knn join processing. InProceedings of the Thirtieth international conference on Very large data bases-Volume 30 2004 Aug 31 (pp. 756-767).

Kalashnikov DV (2013) Super-EGO: fast multi-dimensional similarity join. VLDB J 22(4):561–585