UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Fast Outlier Detection Using Similarity Self-Join Techniques

Eugênio Ferreira Cabral

Dissertação de Mestrado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP
SÃO CARLOS

Eugênio Ferreira Cabral

Fast Outlier Detection Using Similarity Self-Join Techniques

Dissertation submitted to the Institute of Mathematics and Computer Sciences – ICMC-USP – in accordance with the requirements of the Computer and Mathematical Sciences Graduate Program, for the degree of Master in Science. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Robson Leonardo Ferreira Cordeiro

USP – São Carlos
April 2021

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

Ferreira Cabral, Eugênio
F117f Fast Outlier Detection Using Similarity Self-Join Techniques / Eugênio Ferreira Cabral; orientador Robson Leonardo Ferreira Cordeiro. -- São Carlos, 2021.
99 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2021.

1. outlier detection. 2. similarity self-join. I. Leonardo Ferreira Cordeiro, Robson, orient. II. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

Eugênio Ferreira Cabral

Detecção Rápida de Casos de Exceção Utilizando Técnicas de Auto-Junção por Similaridade

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Robson Leonardo Ferreira Cordeiro

USP – São Carlos
Abril de 2021

To those who inspired it and will not read it.

ACKNOWLEDGEMENTS

Special thanks and love to my family, who always valued my education and supported me continuously throughout this project with encouragement and patience. All my efforts are dedicated to them.

To my dear friend Vinícius Ferreira da Silva – thank you for your support, kindness, and invaluable friendship. A very special thanks to Isabela Martins for her incredible patience and motivation through this experience.

Thanks to all members of GBDI, in special those who gave me the opportunity to learn from their incredible minds; Cézanne Alves, Gabriel Spadon, Jadson José, Lucas Scabora, Willian Dener - in alphabetical order.

To my advisor Robson Cordeiro for his trust, dedication, support and great guidance in the development of this project.

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

“We have to remember that what we observe is not nature in itself,
but nature exposed to our method of questioning.”
(Werner Karl Heisenberg)

RESUMO

CABRAL, E. F. **Detecção Rápida de Casos de Exceção Utilizando Técnicas de Auto-Junção por Similaridade**. 2021. 99 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2021.

A democratização dos dispositivos eletrônicos ao longo dos anos incentivou indivíduos e indústrias a produzirem dados a um baixo custo. Como consequência, a produção de dados aumentou globalmente em ritmo acelerado. Com essa produção de dados cada vez maior, as indústrias exigiram melhores ferramentas para encontrar padrões e melhorar seus processos de tomada de decisão. Alguns eventos em particular podem não encaixar em nenhum padrão e ainda assim trazerem informações importantes. São usualmente eventos raros que não correspondem à maioria dos dados, também conhecidos como anomalias, exceções ou *outliers*. Eles podem representar falhas, fraudes, invasões ou condições anormais em sistemas. Detectar esses eventos o quanto antes é crucial em aplicações reais, como finanças, redes sociais e controle de qualidade. Vários algoritmos fornecem excelentes resultados em termos de qualidade, porém na prática, se mostram ineficientes para lidar com dados volumosos. Abordagens mais eficientes pressupõem que uma exceção pode ser identificada buscando por instâncias similares, também conhecidas como vizinhas devido à proximidade espacial entre as instâncias. As estruturas de dados armazenam dados e realizam sucessivas operações de busca por vizinhança para obter informações sobre a densidade da vizinhança, a qual é usada na detecção de exceções. Essa operação tem sido muito pesquisada na comunidade de busca por similaridade ao longo dos anos. Nessa comunidade, é sabido que essas sucessivas operações podem ser substituídas por uma junção por similaridade, mas essa observação não parece óbvia na literatura de detecção de casos de exceção porque praticamente todos algoritmos criam suas próprias estratégias de busca por similaridade. A junção por similaridade é uma operação que, dado dois conjuntos de dados e um limite de similaridade, o objetivo é encontrar todos os pares de instâncias similares. Porém, quando apenas um conjunto de dados é fornecido, essa operação é denominada auto-junção por similaridade. Os algoritmos para essa operação visam melhorar a eficiência em uma ampla gama de aplicações. Como casos de exceção são eventos raros e divergentes da maioria, instâncias com poucos pares podem ser uma exceção. Neste trabalho, propomos investigar como essa sobreposição de conceitos pode ser benéfica para melhorar o desempenho e a escalabilidade de algoritmos de detecção de exceção. Propomos dois novos algoritmos baseados em técnicas de junção por similaridade - ODSSJ e HySortOD. Os resultados experimentais sugerem que as soluções são 3 *ordens de magnitude mais rápida* que os algoritmos estado da arte existentes.

Palavras-chave: detecção de casos de exceção, junção por similaridade.

ABSTRACT

CABRAL, E. F. **Fast Outlier Detection Using Similarity Self-Join Techniques**. 2021. 99 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2021.

The democratization of electronic devices over the years encouraged individuals and industries to produce data at a cheap price. As a consequence of this phenomenon, the data production increased globally at a fast pace. With this ever-growing data production, industries demanded better tools to find patterns in the large volume of data available and improve their decision-making processes. Some particular events might not fit in any existing pattern and yet bring important business insights. They are usually rare events that do not conform with the majority of the data, often classified as anomalies, exceptions or outliers. They can represent failures, frauds, scamming, invasions or abnormal conditions in systems. Detecting this type of event as soon as possible is crucial for real-world applications such as in finance, healthcare, social networks and quality control. Several algorithms have been introduced in the literature providing outstanding results in terms of effectivity, but, in practice, the existing alternatives are still very much expensive in terms of runtime. The most efficient approaches assume that an outlier can be identified by searching for similar instances, also known as neighbors due to their close proximity in the feature space. Data structures are used to store the instances and perform successive neighborhood search operations as a way to take advantage of neighborhood properties and detect outliers. Such type of operation has been strongly researched in the community of similarity search over the years. It is well-known by this community that successive neighborhood searches can be replaced by a single similarity join operation, but this observation does not seem obvious in the outlier detection literature because virtually all algorithms develop their own strategies to search for similar instances. Similarity join is a fundamental operation in database systems; given two datasets and a similarity threshold, the goal is to find all pairs of similar instances. When only one dataset is given, this operation is named similarity self-join; it returns a set of similar pairs for each instance. In this context, since outliers are rare events and diverge from the majority, the instances with few similar pairs are potential outliers. Many join-based algorithms aim to improve the efficiency of the operation in a diverse range of applications. In this work, we investigate how this overlap of concepts can improve the runtime and scalability of outlier detection algorithms. We propose two novel outlier detection algorithms that use join-based techniques - ODSSJ and HySortOD. Our experimental results suggests that our solutions are 3 *orders of magnitude faster* than existing state-of-the-art algorithms.

Keywords: outlier detection, similarity join.

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF TABLES

LIST OF SYMBOLS

ε — Range search radius

k — Number of neighbors in kNN

$f(\ldots,\ldots)$ — Distance or similarity function

$rng(\ldots,\ldots,\ldots)$ — Range search function

q — Query center instance

τ — Threshold for minimum number of neighbors to be flagged as outlier

X — Array of numerical instances

m — Number of instances in X

d — Number of dimensions

p — Array index of X

j — Column index

x_p — p-th instance in X

$x_{p,j}$ — p-th index in dimension j

C — Array of counts

c_i — Number of instances in the neighborhood of i-th object in C; where object means instance (ODSSJ) or hypercube (HySortOD)

O — Array of outlier output; where outlier output means flag (ODSSJ) or score (HySortOD)

o_p — p-th outlier output in O

t — Join threshold value for Super-EGO

b — Number of bins

l — Hypercube side size (or length)

H — Array of hypercubes

n — Number of hypercubes in H

i — Array index of H

h_i — i-th hypercube in H

h_k — k-th hypercube neighbor

$h_{i,j}$ — i-th index in dimension j

W — Array of densities

w_i — Density value for i-th hypercube

w_{max} — Maximum density value

$N(h_i)$ — Set of neighbors for hypercube h_i

$MinSplit$ — Minimum threshold value for creating node branches

P — Parent node

P_{value} — Value stored in parent node

P_{begin} — First mapped index in the parent node

P_{end} — Last mapped index in the parent node

CONTENTS

CHAPTER

1

INTRODUCTION

1.1 Context

The increasing volume of data produced in today's world offers challenges in many areas, especially for databases and data analysis (CHE; SAFRAN; PENG, 2013). Database systems have undergone a dramatic evolution in recent decades to accommodate several industry needs due to the large volume of data, and the data analysis has also played a vital role in the decision-making and optimization process. As a consequence of this evolution, many companies have adopted the data-informed culture to guide their decisions to gain competitive advantage and offer better products and services (STRIPHAS, 2015).

To achieve the benefits of this culture, the specialists must ask the business questions, and their answers must be obtained and supported by data. The process of searching for particular events or patterns in data is known as data mining, and several algorithms have been proposed to facilitate this process. An example of a question that could be answered through data mining would be *Who are the users who commit fraud?* and *What are their characteristics?* Since fraudulent actions are rare events with particular behaviors that diverge from the majority, such actions can be seen as exceptions, anomalies or outliers; therefore, outlier detection algorithms could be employed in this context to find the expected answer. Another example of a question that could be answered by outlier detection is *What are the defective products in an industrial production line?* Most items in a production line usually meet the quality standard of the business, but rare problematic events can cause defects or malfunctioning in the production machinery and consequently the production of defective products. Detecting such cases can help the operation staff to develop diagnostics to improve the quality control and minimize future material waste. The detection of these cases allows companies to plan preventive actions or improve their processes so that they can offer more appropriate services. As the industry increases its data production, it becomes manually impractical to detect rare events that can provide new insights or understand unexpected behaviors in systems. In this context of a large volume of data and the crucial task of outlier detection, our work aims to propose more efficient and scalable solutions.

1.2 Problem and Motivation

The research on outlier detection has its origins a few decades ago when statistical methods were employed, and a primary requirement was to assume the underlying data distribution (HAWKINS, 1980). However, to spot the correct data distribution is not always trivial in most real-world applications, especially in this constantly changing world. To work around this issue, part of the research community started to interpret the detection of outliers as a search problem; then, a distance-based notion was adopted and required to perform searches for every instance using a distance function, e.g., Euclidean distance, without the need to assume the data distribution (KNORR; NG, 1998; BREUNIG *et al.*, 2000). The neighborhood concept is the basis of these searches, which assumes that instances that are close to each other in the feature space are considered neighbors, i.e., similar instances. The intuition behind this approach is that "normal" instances would have several neighbors, while the outliers would be the rare cases with few neighbors.

The neighborhood concept can have two common interpretations; for example, given a random instance, it is possible to define the k-nearest neighbors (kNN) or all instances that are within a ε distance (ε-neighborhood). These two definitions inspired several algorithms to achieve high-quality results (GOLDSTEIN; UCHIDA, 2016; CAMPOS *et al.*, 2016) by only performing successive neighborhood search operations, one operation per instance, and then evaluating the instance neighborhood based on some criterion. However, they often strive to provide scalable solutions (ORAIR *et al.*, 2010; GOLDSTEIN; UCHIDA, 2016; KIRNER; SCHUBERT; ZIMEK, 2017). It is well known by the similarity search community that successive neighborhood search operations can be replaced by one single similarity join operation, which is a fundamental operation in database systems (SILVA; AREF; ALI, 2010). In this context, a ε-join is equivalent to successive ε-neighborhood search operations, and a k-nearest neighbors join (kNN-join) is equivalent to successive kNN search operations, where ε-join and kNN-join are particular types of similarity join. These types of joins have been a popular topic of research by the similarity search community for many years in a wide range of applications, mainly focused on the analysis of large volumes of data.

In this work, we are interested in taking advantage of the techniques employed by similarity join algorithms to improve the efficiency of the outlier detection task. We first demonstrate how these two tasks are related and how to adapt an existing join-based algorithm to detect outliers. As a byproduct of our adaptation, we propose a novel algorithm named ODSSJ. Then, we revisit the hypercube-based notion for outlier detection and combine it with a sorting strategy employed by join-based algorithms to propose a novel algorithm named HySortOD that can scale to large volumes of data. To the best of our knowledge, the link between outlier detection and similarity self-join has not been explicitly investigated previously in the literature. This observation and the need for scalable solutions are the sources of motivation for our hypothesis that aims to improve the performance and the scalability of outlier detection algorithms.

Hypothesis: The use of similarity self-join techniques makes it possible to detect outliers more efficiently.

1.3 Contributions

This work proposes two novel algorithmic approaches based on the aforementioned hypothesis. We conducted extensive experiments using several real-world datasets from different domains, like Chemistry, Biology, Healthcare, Remote Sensing and Networking. According to our experimental results, we highlight the following contributions:

C1 **Generality**: Our proposed algorithms are designed based on the well-known neighborhood assumption that provides flexibility to be used in a wide range of applications.

C2 **Simplicity**: Our approach does not require labels to learn how to detect outliers and parameter intuitiveness allow our algorithms to be used straightforwardly in most real-world scenarios.

C3 **Interpretability**: The neighborhood-based approach allows specialists to easily interpret and disseminate the results.

C4 **Speed**: We take advantage of similarity join techniques to avoid unnecessary computations, and show that this approach is a powerful tool to reduce the outlier detection runtime.

1.4 Organization

The next chapters of this work are organized as follows. Chapter 2 provides the fundamental concepts that are used as basis for this work regarding outlier detection and similarity join. Chapter 3 summarizes the relevant existing work in the literature that inspires this present work. Chapter 4 discusses about the relationship between similarity self-join and outlier detection; it also introduces the mechanisms and experiments of our join-based algorithm named ODSSJ. Chapter 5 introduces our novel sorted hypercube algorithm named HySortOD and presents the experiments that demonstrate its efficiency. Finally, in Chapter 6, we provide the concluding remarks and highlight the contributions of this work with respect to our findings.

CHAPTER

2

FUNDAMENTAL CONCEPTS

2.1 Outlier Detection

The detection of outliers has been researched in many areas over the years, and several applications benefit from the algorithms developed by this research, such as in quality control (JAUHRI; MCDANEL; CONNOR, 2015), sensor networks (SHAHID; NAQVI; QAISAR, 2015), cyber intrusion (JABEZ; MUTHUKUMAR, 2015), healthcare (ANBARASI; DHIVYA, 2017), social networks (BINDU; THILAGAM; AHUJA, 2017) and finance (TRIPATHI *et al.*, 2018). As a consequence of such a variety of applications, alternative names like abnormalities, discordant, deviants or anomalies emerged to contextualize with the application domain. However, in general, they serve the same purpose: to identify instances that are significantly different from the others. In Ayadi *et al.* (2017, p. 3), the authors list several attempts to define an outlier precisely, but the consensus is not clear.

Two of the first known definitions state that "an outlier is an observation which deviates so much from the other observations as to arouse suspicions that it was generated by a different mechanism" (HAWKINS, 1980), and "an observation (or subset of observations) which appears to be inconsistent with the remainder of that set of data" (BARNETT; LEWIS, 1994). In these examples, the challenge is the interpretation of "deviates so much" and "appears to be inconsistent", because they are both vague definitions that leave room for discussion. In order to deal with this issue, the alternative is to make assumptions based on the target application.

A high-level definition concerning the degree of outlierness of an instance can be helpful to consider while tackling an outlier detection problem; it is known as *outlierness score* (AGGARWAL, 2017, p. 3). This score is usually a continuous value in the $[0, 1]$ interval where high values represent a high degree of an instance being an outlier, and vice versa as it is illustrated in Figure 1. The interval can be intuitively divided into three instance types:

- Inliers describe what is considered to be a normal behavior for the application;
- Noise represent the semantic boundary between what is normal and the outliers;
- Outliers are considered significantly different from the normal behavior.

In some algorithms, noise instances are usually reported as outliers because they might provide significant insights for the specialist. The threshold values (or sub-intervals) for each of the previous types depend on the application. In practical terms, every application imposes some limitations, and to find a general definition for outliers that copes with all scenarios might not be a straightforward task. Therefore, to understand the application goals and constraints is crucial in the outlier detection problem.

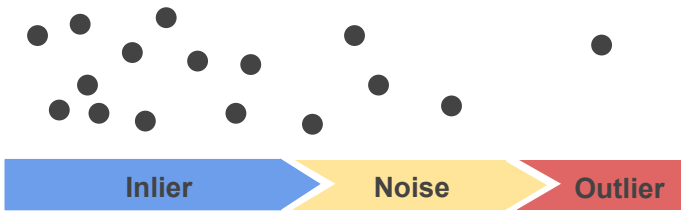


Figure 1 – Illustration of instances scattered in a 2-dimensional space (top) with their respective type (bottom). Inliers indicate the lowest outlierness scores while outliers indicate the highest ones; noise instances are in between inliers and outliers.

Part of these limitations is associated with the dataset that is used as input. Figure 2 illustrates an overview of the basic structure of the input data. The datasets for outlier detection consist of a collection of instances (also known as observations, objects, tuples or vectors) with one dimension (unidimensional) or more (multidimensional). The terms dimension, attribute, feature and variable are often used interchangeably in the literature. Dependency between instances might exist in problems that have sequential data, such as audio or time series, where a given instance depends explicitly or implicitly from the previous instance. To develop outlier detection algorithms, one must consider not only the instance dependency, whenever it exists, but the data type of each dimension as well. Each dimension can be either numerical (quantitative) or categorical (qualitative), for which different metrics can be used to distinguish instances in the space. For example, numerical dimensions often use Minkowski distance functions (LESOT; RIFQI; BENHADDA, 2009) for this task, although other distance functions can also be employed. Categorical dimensions commonly require a semantically meaningful distance function that is constructed for each application. Also, there might exist a special type of dimension that indicates the label of each instance; it specifies whether a given instance is an outlier or not. However, not all datasets provide labels for any instance, and the availability of this information changes the strategy of how the algorithm must learn about the problem.

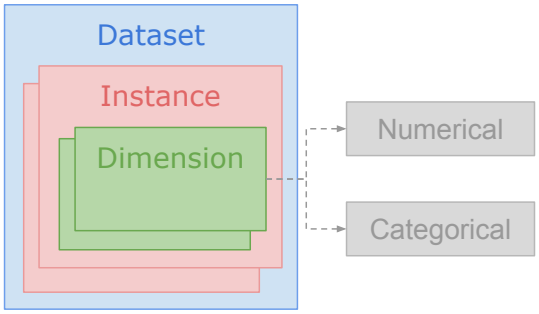


Figure 2 – The structure of an input dataset. A dataset contains a set of instances, where each instance has a set of dimensions that can be either numerical or categorical.

Figure 3 illustrates the most common learning schemes that can be employed depending on the input data. They are briefly described in the following:

- Unsupervised Learning is when the algorithm does not require the label dimension and the learning relies only on the knowledge extracted from the other dimensions;
- Supervised Learning takes advantage of the label dimension and learns by example to distinguish future unlabeled instances in outliers and inliers;
- Semi-Supervised Learning is when only a few (not all) instances are labeled as outliers, then, the algorithm can mix strategies from the previous two learning schemes to distinguish future unlabeled instances in outliers and inliers.

Additionally, these learning schemes must consider the mode of analysis, between *offline* and *online* detection. The offline mode is when there is one static dataset available, while the online mode considers a stream of data. Both cases impose different algorithmic strategies and limitations that should be treated during the algorithm design and application.

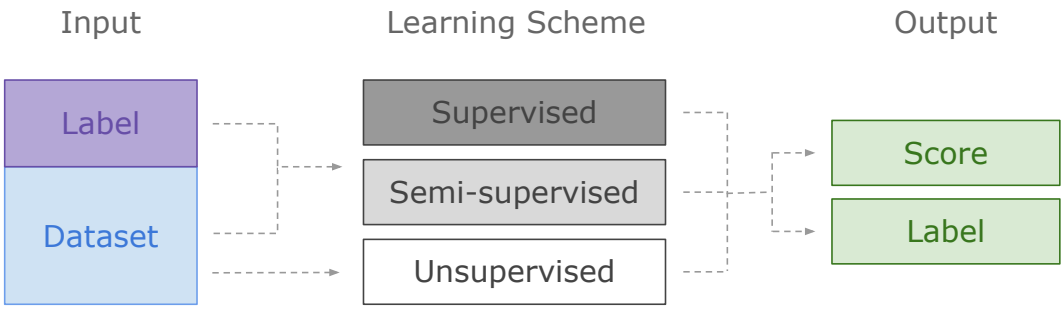
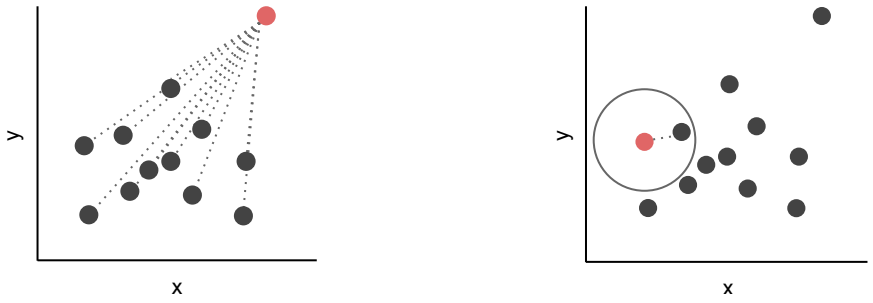


Figure 3 – Illustrates the pipeline components for the task of outlier detection: input, learning scheme and output. The input dataset (left) can optionally contain labels, and it is used in the proper learning scheme (center) to finally report the output (right) that can be either a score or a label.

For any learning scheme, it is also important to define beforehand the type of outlier to look for. In this work, we focus on unsupervised learning scheme and proximity-based outliers due to its conceptual affinity with similarity join; we provide an in-depth analysis in such affinity in Section 4.1. According to Aggarwal (2017) proximity-based outliers can be distinguished as global and local outliers, as it is described in the following and illustrated in Figure 4.

- Global outlier (or point outlier) is when an instance deviates from the rest of the dataset. For example, when errors or malfunctioning is recorded in data, such instances are considered unusual compared with other instances, as it is illustrated in Figure 4a.
- Local outlier is when an instance has few neighbors relative to its surroundings, but not necessarily from the rest of the dataset. For example, identifying businesses that are out of a certain radius in a neighborhood, as it is illustrated in Figure 4b.



(a) Global outlier (b) Local outlier

Figure 4 – Illustration of global and local outliers. In Figure 4a, the outlying instance is far from most instances in the dataset, while, in Figure 4b, the outlier has only one instance in its surroundings.

The main challenge for both the global and the local types is to define how far from the other instances a given instance must be to be considered an outlier. Note that, despite the aforementioned differences among types of outliers, some algorithms can detect multiple types, and the interpretation of the results is commonly up to the specialist.

Aside from the algorithm and its applicability, there are two ways to indicate whether an instance is an outlier or not: it can be done either by using a score or by using a label, as it is illustrated in Figure 3. Scores are continuous values that quantify the *outlierness* of an instance. In other words, they describe the instance's degree of outlierness. They can express distances, probabilities or some relative measure in the interval $[0, 1]$. On the other hand, labels are binary values stored in categorical attributes to indicate whether or not an instance is an outlier. Furthermore, a score value can be converted to a label if a threshold value is set. Thus, when the score value is higher than the threshold, the corresponding instance is labeled as an outlier; otherwise, it is labeled as not an outlier, i.e., an inlier or noise.

The aforementioned concepts about outliers compose the essential building blocks to design most outlier detection algorithms. Since this MSc work focuses on unsupervised learning

algorithms, let us introduce in the following some common strategies found in the literature, such as Extreme Value Analysis, Probabilistic models and Proximity-based algorithms.

Extreme Value Analysis (PICKANDS, 1975) assumes that the dataset values that are either too small or too large should be flagged as outliers. Thus, the main idea here is to find *statistical tails* of the distribution. Despite being intended for unidimensional problems, it can also be applied to multidimensional ones. For example, Leys *et al.* (2018) developed a simple approach to handle outliers in multidimensional datasets with a small sample size from the psychology domain. However, this algorithm should not be used as a generic outlier detection tool because it is designed to detect specific types of outliers, i.e., global outliers, and it is also incapable of identifying outliers in sparse interior regions of a dataset. Nevertheless, it can be used as a final step in algorithms that quantify the deviations or distances of instances in the form of a numerical score. For those cases, the usage of Extreme Value Analysis can be effective.

Probabilistic models rely on a closed-form probability distribution, and the parameters of the probability models must be learned from the dataset. The assumption for these models depends on the choice of data distribution. Whenever all mixture components have a generative model available, this approach can be applied in any data type. However, the major challenge here is to find the appropriate distribution for the dataset. For unidimensional problems, to identify the distribution might not be so challenging. However, to select the appropriate distribution for multidimensional cases is not a simple process, and overfitting can become a problem as the number of parameters increases. Depending on the model, the parameters cannot be easily interpreted by a specialist, and it becomes an issue if a diagnostic is required to justify why an instance was flagged as an outlier. Like it happens with the Extreme Value Analysis, the Probabilistic models can also be used as a final step in outlier detection algorithms. For example, to convert outlierness scores into probability estimates can be a powerful strategy to provide high-grade result interpretation (GAO; TAN, 2006).

The proximity-based approach is often a source of inspiration for many algorithms. Its popularity is mainly due to the simplicity of implementation and the easiness of result interpretation. The notion of proximity assumes as outliers the instances that are isolated from the remaining ones. There are many variations of this approach and we provide as follows the details for three common alternatives: Clustering-based (Section 2.1.1), Angle-based (Section 2.1.2) and Neighborhood-based (Section 2.1.3). Note that there are other proximity-based ways to detect outliers, such as Linear Models (MA; PERKINS, 2004), Information Theoretic Models (WU; WANG, 2013) and Spectral Models (SATHE; AGGARWAL, 2016), but these are not covered in this monograph since they are often used in very particular contexts.

2.1.1 Clustering-based Outlier Detection

Clustering algorithms share a complementary relationship with outlier detection algorithms. The clustering process groups similar (or dense) instances from a dataset into clusters, i.e., subsets of similar instances, while outlier detection identifies instances that do not conform with any of these clusters. Many clustering algorithms detect outliers as a byproduct of their analyses. However, the non-membership of an instance regarding all clusters can be noise rather than an outlier because the clustering algorithm is not necessarily measuring the deviation level from regular instances, but rather its (dis-)similarities with other instances. On the other hand, once the clusters are created the detection of outlying instances tends to be faster because the algorithm must compare whether the new instance belongs to the cluster or not. Instead of comparing the new instance against all cluster instances, it can be done by just checking the distance from the cluster centroid or whether the instance lies within the cluster boundaries. Figure 5 shows an example of this approach, where the red instance is compared against two cluster centers rather than comparing it with the instances that are clustered.

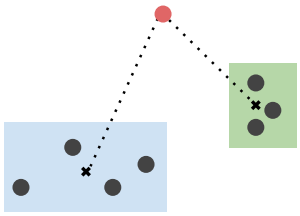


Figure 5 – Example of clustering-based outlier detection. Illustration of two clusters and an outlier instance that does not belong to any cluster.

According to Chandola, Banerjee and Kumar (2009), there are three categories of clustering-based outlier detection algorithms that are distinguished by their assumptions. The first one assumes that "regular instances belong to a cluster, while outliers do no belong to any cluster". A classical algorithm that implements such an assumption is DBSCAN (ESTER *et al.*, 1996). When an instance cannot be reached by other instances from a maximum distance ε or it does not have enough instances (*MinPts*) in its ε-neighborhood, then this instance does not belong to any cluster. Therefore, it is flagged as an outlier. GLOSH (CAMPELLO *et al.*, 2015) was developed based on the same clustering principles, but focused specifically on outlier detection. The second category of algorithms assumes that "regular instances lie close to their closest cluster centroid, while outliers are far away from their closest cluster centroid". This assumption can be implemented with KMeans (LLOYD, 1982) by flagging as outliers instances that exceed a distance threshold α from their closest cluster centroid. Finally, the third category assumes that "regular instances belong to large and dense clusters, while outliers belong either to small or sparse clusters". For example, CBLOF (HE; XU; DENG, 2003) spots outliers by measuring the probability of an instance to belong to a given cluster with respect to its distance to the cluster and the cluster size.

To summarize, different assumptions allow a clustering algorithm to spot outliers as a byproduct of its process, and these assumptions must be considered beforehand according to the desired types of outliers. All three approaches allow a clustering algorithm to detect outliers, but they are not optimized for outlier detection in terms of effectivity and efficiency. Besides that, there are algorithms such as KMeans that force all instances to belong to a cluster, which is not desirable for outlier detection. Instead, algorithms such as DBSCAN can produce more interesting results because not every instance has an assigned cluster. Most clustering algorithms can operate in an unsupervised mode and can be adapted to handle mixed data types.

2.1.2 Angle-based Outlier Detection

Another interesting approach to detect outliers is based on the divergence in the direction of instances relative to one another. It assumes that instances at the boundaries of the feature space are likely to enclose the entire dataset within a small angle, whereas instances in the interior are likely to have other instances around them at very different angles. In other words, it is expected that inliers have a large variation of angles and outliers a small variation of angles. In Figure 6, the highlighted instance illustrates the difference between angle variation in inlier and outlier instance. When comparing angles with two pairs of other instances, it is possible to note that the outlier instance encloses several other instances, and the angle variation is low in contrast to those instances that are in the interior of the feature space.

(a) Angle-based inlier (b) Angle-based outlier

Figure 6 – Illustration of angle-based inlier and outlier. In Figure 6a, the highlighted instance has a large angle variation indicating an inlier instance, while, in Figure 6b the highlighted instance encloses the entire dataset within a small angle variation indicating an outlier instance.

The first algorithm to implement this idea is ABOD (KRIEGEL; SCHUBERT; ZIMEK, 2008). The authors introduce a metric named Angle-Based Outlier Factor (ABOF) that describes the variance over the angles between the difference vectors from an instance to all possible pairs of other instances in the dataset weighted by the instances distances. This strategy arguably provides better results for high-dimensional datasets because the variance of angles does not deteriorate as the number of dimensions increase, differently from what happens to distance measurements. However, the proposed metric requires a triple of instances to determine the angle. Thus, to perform the angle computation of all instances, the algorithm requires a cubic

time complexity to assign the ABOF value for each instance. In order to tackle this issue, the authors proposed the FastABOD algorithm, which is based on samples of the dataset considering the calculation of angles only with the k-nearest neighbors of each instance. This approximation improves the time complexity of the original algorithm significantly, but the accuracy of results depends on the value of k, and, according to the original experiments, the results become worse as the data dimensionality increases. Besides, the authors also provide a conservative (lower bound) approximation of ABOF, named LB-ABOF, that reduces to quadratic the time complexity, but the quality does not necessarily improve. Other algorithms explore this angle-based idea to reduce even more the time complexity. FastVOA (PHAM; PAGH, 2012) proposes a near-linear approximation based on the ABOD algorithm; it introduces the metric Variation Of Angles (VOA) as an alternative to ABOF without the normalization factor. The authors argue that ABOF is normalized by the distances between instances, and, in high-dimensional spaces, it becomes less meaningful due to the *curse of high dimensionality* (BELLMAN, 1961).

Angle-based algorithms are indeed an original idea in outlier analysis. Nevertheless, their assumption is not applicable in many datasets, because outliers can also lie in the interior of the feature space. In contrast, the neighborhood-based algorithms aim to explore these interior cases in depth. Furthermore, the trade-off between time complexity and accuracy is a challenging problem to solve in the angle-based approach.

2.1.3 Neighborhood-based Outlier Detection

The notion of neighborhood is based on the similarity between instances, that is, instances in a given neighborhood are considered to be similar to each other due to their closeness in the feature space. It requires a distance-based similarity function to distinguish between neighboring and non-neighboring instances, besides a comparison criterion between neighborhoods to detect outliers. This criterion is usually a density measure that considers ratios between the density around an instance of interest and the density around its closest neighbors. The neighborhood of an instance can be defined as ε-neighborhood or k-nearest neighborhood (kNN).

(a) ε-neighborhood-based outlier (b) kNN-based outlier

Figure 7 – Illustration of neighborhood-based outlier detection. The outlying instance in Figure 7a has no neighbors in the ε radius, while, in Figure 7b, the 2 nearest neighbors are far away.

As it is illustrated in Figure 7a, the ε-neighborhood is equivalent to a range query with a given query instance as the center, and all neighboring instances are at most ε distant from the center. The kNN defines as neighbors the k-nearest instances. Figure 7b shows an example of a 2NN query. Moreover, in Figure 7 we can observe that in the same data distribution the ε-neighborhood (Figure 7a) does not identify any neighbors for the red instance, but in the 2NN strategy (Figure 7b) identifies two neighbors for the red instance. Based on this observation, it is crucial to notice that, in some situations, the kNN might find that its nearest neighbors are too far away and such cases must be considered when designing algorithms based on this strategy. A more general view of these definitions is that, for a given instance, the distance to its k^{th} nearest neighbor is equivalent to the radius ε of a hypersphere centered at the given instance that contains k other instances (CHANDOLA; BANERJEE; KUMAR, 2009, p. 24). Since the search process is performed in a local region, i.e., it is either limited by ε or by k, these algorithms are known as local outlier detectors. The similarities among the definitions allow the concept to be generalized. There is a generalized view of several algorithms in the literature to allow a clear theoretical comparison among the existing algorithms (SCHUBERT; ZIMEK; KRIEGEL, 2014).

To the best of our knowledge, the first algorithm to use a distance-based approach was DB-Out (KNORR; NG, 1998). The idea is to provide an outlierness score for a given query instance by counting the number of nearest neighbors that are at no more than a ε distance from the query instance. This approach can only handle numerical data. As an alternative, algorithm CNB (LI; LEE; LANG, 2007) introduced a common-neighbor-based distance function to measure the similarity between categorical instances. Nevertheless, real-world datasets contain missing values, and specific distance functions are required to meet this need. Thus, algorithm HOT (WEI *et al.*, 2003) uses the connectivity property to replace distance metrics so to better handle missing values and mixed data types.

LOF (BREUNIG *et al.*, 2000) further explores the idea of counting instances in a ε-neighborhood to compute the neighborhood density. It introduces the Local Outlier Factor (LOF), which assigns an outlierness score for each instance. The algorithm compares the local, ε-neighborhood-based density of each instance with the corresponding neighborhood density of each of its k nearest neighbors; instances with high LOF value are considered to be outliers. However, to determine an appropriate value for parameter k is not a trivial task. Therefore, LOCI (PAPADIMITRIOU *et al.*, 2003) introduced the concept known as Multi-granularity DEviation Factor (MDEF). In this approach, the main difference from LOF is that instead of comparing local densities of an instance with those of its k nearest neighbors, it compares with those instances in a ε-neighborhood. Although the approaches are substantially similar in the theoretical sense, LOCI's implementation introduces several challenges in time complexity.

The algorithms in this category usually perform a comprehensive analysis in the dataset to find outliers, which can lead to high computational costs. Thus, several algorithms employ different strategies to improve efficiency. DB-Out uses a hypercube-based approach to discretize and summarize the space so to reduce the number of instances to be analyzed. RBRP (GHOTING; PARTHASARATHY; OTEY, 2008) uses a recursive approach that iteratively partitions instances into k bins with a fixed number of iterations. Then, it scans through the bins to find the approximate nearest neighbors of each instance. When the number of nearest neighbors does not exceed a threshold, then the instance is flagged as an outlier. This strategy allows the specialist to retrieve the top n outliers by approximating the computation and consequently speeding-up the outlier detection process.

One of the main advantages of the neighborhood-based algorithms is that no prior knowledge about the data distribution is required. As long as it is provided a distance function that is well suited to compare the instances of a dataset of interest, this approach can be employed. Also, the level of interpretability that this approach provides is a positive aspect; even approximate solutions deliver a high level of detail. This characteristic allows many specialists to better understand outliers in real applications. However, the major drawback of this approach is the computational cost to detect outliers. A naïve implementation requires that n instances must be compared against to $n-1$ instances leading to a $O(n^2)$ time complexity, which significantly increases the computational cost of this approach. Thus, neighborhood-based algorithms usually need to address this issue to become suitable for real-world large datasets.

2.2 Similarity Join

The Similarity Join (SJ) operation is widely used in data mining. For instance, it has already been used in entity resolution (CHEN; KALASHNIKOV; MEHROTRA, 2009), near duplicate document identification (WANG *et al.*, 2011), gene sequences comparison (WANDELT *et al.*, 2013), query autocompletion (ISHIKAWA *et al.*, 2013), document clustering (LIN; JIANG; LEE, 2014), record linkage (ADHAV; KUMAR, 2015) and data cleaning (GIANNAKOPOULOU *et al.*, 2017). For clarity, this section describes the similarity join in terms multiple executions of a simpler operation, i.e., the Similarity Search (SS). For both operations, in order to determine whether any two instances x and y are similar to each other, two components must be defined beforehand: a similarity function $f(x,y)$ and a similarity condition.

The similarity function returns a continuous score representing how similar two instances are based on the specific rules to handle different data types. For example, the most common functions are the Euclidean distance (LESOT; RIFQI; BENHADDA, 2009) for numerical instances, the LEdit distance (LI *et al.*, 2015) for string instances, and the Jaccard distance (MANN; AUGSTEN; BOUROS, 2016) for categorical/set instances. Some similarity functions have a fundamental property known as *metric* that is useful for many applications. A function is said to be a metric iff the following properties are valid for any possible instance x, y and z:

(1) Symmetry: $f(x,y) = f(y,x)$

(2) Non-negativity: $f(x,y) \geq 0$

(3) Identity: $f(x,y) = 0 \iff x = y$

(4) Triangle inequality: $f(x,y) \leq f(x,z) + f(z,y)$

A similarity condition compares the similarity score with a similarity threshold ε using a logical comparator, such as "less than". When the similarity condition is satisfied, the x and y instances are said to be similar to each other. Thus, a clear understanding of what is said to be similar must align with the requirements of the application. Note that a similarity condition can also be expressed in terms of k instead of ε, for the kNN-based similarity search, but, for brevity, we express it in the following only in terms of ε.

Formal definitions for both the Similarity Search (SS) and the Similarity Join (SJ) operations are provided in the following.

Definition 1 (Similarity Search)**.** Given a dataset X, a query instance q, a similarity function f and a similarity threshold ε, the Similarity Search (SS) returns all instances from X whose distances to instance q are smaller than ε, according to function f. Formally, it is given by:

$$SS_{f,\varepsilon}(X,q) = \{x \mid f(x,q) < \varepsilon;\ x \in X\}$$

Definition 2 (Similarity Join)**.** Given two datasets X and Y, a similarity function f and a similarity threshold ε, the Similarity Join (SJ) returns all pairs of instances whose distances to each other are smaller than ε, according to function f, where the first instance of each pair belongs to X and the second instance belongs to Y. Formally, it is given by:

$$SJ_{f,\varepsilon}(X,Y) = \{(x,y) \mid f(x,y) < \varepsilon;\ x \in X,\ y \in Y\}$$

A particular case of the SJ operation has a central role in this MSc work. Being known as the **Similarity Self-Join (SSJ)** operation, it happens when the datasets X and Y are exactly the same, so there is actually only one dataset to be analyzed. Let us present a formal definition for the SSJ operation in the following.

Definition 3 (Similarity Self-Join)**.** Given a dataset X, a similarity function f and a similarity threshold ε, the Similarity Self-Join (SSJ) returns all pairs of instances from X whose distances to each other are smaller than ε, according to function f. Formally, it is given by:

$$SSJ_{f,\varepsilon}(X) = \{(x,y) \mid f(x,y) < \varepsilon;\ x,\ y \in X\}$$

Based on the previous definitions, one can easily rewrite the definition of the SSJ operation in terms of successive executions of SS operations, which obviously obtain the exact

same result set. Let us express the similar pairs in an equivalent form, where the second element y of the tuple is a set of similar instances instead of a single instance, that is:

$$SSJ_{f,\varepsilon}(X) = \{(x,\ SS_{f,\varepsilon}(X,x)) \mid x \in X\}$$

Although the goal of the SSJ operation is to find all similar pairs of instances, it is also possible to restrict this operation to only find the top-N similar pairs of instances. However, to meet the scope of this work, our interest is to investigate algorithms that find all pairs of similar instances using a metric similarity function and a similarity threshold for the context of numerical instances. Specifically, we are interested in the SSJ operation using the L_2 norm as the similarity function. Formally, it is given by:

$$f(x,y) = ||x\ -\ y||_2 = \sqrt{\sum_{j=1}^{d}(x_j - y_j)^2}$$

In the equation, D is the number of dimensions; x and y are D-dimensional numerical instances, and; x_j and y_j are the j^{th} coordinate values of x and y, respectively.

As it can be seen, the "building blocks" of similarity-based operations are flexible enough to be extended to other applications and goals. For example, in the k-nearest neighbors similarity join (kNN-join) (XIA *et al.*, 2004; DU; LI, 2013), the goal is to join a given query instance with its k-nearest neighbors. This approach is usually faster than the ε-join because the number of instances in the result set is predictable by the value of k. Moreover, there are attempts to estimate the result set size of a ε-join to speed up the operation (LEE; NG; SHIM, 2011). Nevertheless, the SJ concepts can be further explored in matching similar strings (YU *et al.*, 2016), performing SSJ on streaming data (MORALES; GIONIS, 2016), or finding all similar pairs of sets in a collection of sets (JIA *et al.*, 2018).

The simplest way to implement a SJ operation is by creating two nested loops that evaluate the similarity between all instances of datasets X and Y. This strategy is often referred to as brute force, nested loops or naïve search, and its time complexity is $O(n^2)$, which exposes a strong limitation in processing large datasets, becoming not feasible for most real-world applications. Due to the vast applicability of the SJ operation in the literature, researchers proposed novel approaches to improve its runtime so to perform at scale. The major challenge is how to organize the instances in such a way that neighboring instances are stored close to each other so to avoid unnecessary comparisons, since the search cost increases significantly when instances that are far away from each other need to be compared. Thus, several approaches to reduce such complexity have been studied. They are classified into three types, as it is explained in the following sections: Index-based (Section 2.2.1), Hash-based (Section 2.2.2) and Sort-based (Section 2.2.3).

2.2.1 Index-based Similarity Join

One of the earliest attempts to mitigate the runtime of the SJ operation is to index the dataset instances. Brinkhoff, Kriegel and Seeger (1993) presented well-known algorithms to join two existing R-trees considering several strategies to mitigate CPU and I/O costs. However, the traversal cost is high because many leaf nodes need to be visited, and the contribution is limited to data of low dimensionality. To overcome these issues, algorithm ε-kdB-tree (SHIM; SRIKANT; AGRAWAL, 2002) introduced strategies to find appropriate branches in internal nodes and reduce the traversal cost. The authors reported promising experimental results for high-dimensional data. However, the tree construction has a high computational cost, and distributed strategies to join two R-trees are not straightforward, which makes it difficult to process large volumes of data. There is a more recent proposal for *k*NN-join using R-trees in a parallel, distributed environment with MapReduce[1] (DU; LI, 2013). The idea is to partition the dataset into buckets and then build an R-tree for each bucket to find the *k*NN, but the lack of empirical results and the limited scope of the experiments do not suggest significant improvements. Another algorithm that attempts to handle large volumes of data is MR-DSJ (SEIDL; FRIES; BODEN, 2013); it uses a basic index structure (grid) to partition the dataset with MapReduce and many pruning techniques to speed up the SSJ operation, but a significant drawback is the intense network communication. All of these proposals attempt to mitigate the runtime cost by providing efficient strategies in contexts where indexing can be advantageous.

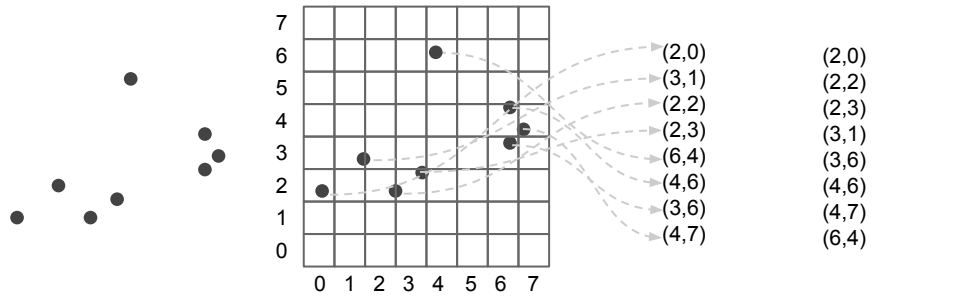
2.2.2 Hash-based Similarity Join

Another approach to organize instances is employed by the hash-based algorithms. The idea is to use a hash function to allocate instances to partitions and then perform the join on pairs of partitions in a recursive fashion. The first algorithm to employ hashing for similarity join is SHJ (LO; RAVISHANKAR, 1996); it consists of a pool of buckets, an assignment function, and a recursive bucket-join step. The main advantage of this approach is that no index is required, but often the buckets end up with unbalanced numbers of instances, and the join performance is compromised. The most successful employment of this approach uses MapReduce (SILVA; REED, 2012; SARMA; HE; CHAUDHURI, 2014; MCCAULEY; MIKKELSEN; PAGH, 2018). Some algorithms for multidimensional data may suffer from a high replication rate. Thus, an additional step is required to eliminate the duplicates. Also, these algorithms may perform network-intensive communication to coordinate the join process. The recent algorithm C2Net (LI; SHAO; FU, 2018) shows advancements in solving this issue by considering a collision counting strategy based on locality-sensitive hashing. In contrast to index-based algorithms, it allows the SJ to operate at scale by distributing the instances across several nodes without the additional cost of maintaining an index structure.

[1] MapReduce is a programming model to process large datasets with a parallel, distributed algorithm running on large clusters of commodity machines.

2.2.3 Sort-based Similarity Join

An alternative approach to organize data in such a way that neighboring instances are stored close to each other is to use a sorting strategy. Several algorithms apply different sorting criteria to search for neighbors more efficiently, and the main advantage is that no advanced data structure is required because these approaches exploit the instances' ordering properties. ZC[2] (ORENSTEIN, 1991) is one of the first algorithms to use the sort-based approach. It takes advantage of the space-filling z-curve to compress the space into a single dimension and performs the sorting based on the values of this dimension. A similar algorithm is MSJ (KOUDAS; SEVCIK, 2000), where the data is sorted with respect to the space-filling Hilbert curve. Another algorithm that combines space-filling curves and ordering is LSS (LIEBERMAN; SANKARANARAYANAN; SAMET, 2008); it casts a SJ operation as a GPU sort-and-search problem, where a set of shifted space-filling curves is used to enclose several instances and narrow down the search space. The main problem with approaches based on space-filling curves is that, for data of moderate-to-high dimensionality, some neighboring instances may be stored far away from each other, which makes the neighborhood property irrelevant. Thus, to tackle this dimensionality issue, GESS (DITTRICH; SEEGER, 2001) introduced a sorting criterion based on the lexicographical order for numerical datasets. A similar idea is used in GORDER (XIA *et al.*, 2004), which sorts grid cells lexicographically and allows the dataset to be partitioned to improve the join execution. Both algorithms employ different strategies to overcome the high computational cost associated with the join operation, but the main difference between them is that GORDER proposes a kNN-based solution while GESS uses a ε-neighborhood-based solution.



(a) Random instances (b) Virtual grid (c) EGO-sorted cells

Figure 8 – Example of the EGO-sort procedure

The idea of lexicographical ordering is investigated in depth by the Epsilon Grid Order (EGO) family of algorithms, which is based on the ε-neighborhood variant of similarity join. The first of such algorithms is EGO-Join (BÖHM *et al.*, 2001). It provides strategies to improve the runtime by prioritizing relevant dimensions; the main idea is to lay an equi-length grid with cells of length ε over the data space and sort the grid cells lexicographically, i.e., to perform an operation named by the authors as EGO-sort, thus allowing the algorithm to load chucks of

[2] The algorithm is named after the Z-Curve.

neighboring instances in main memory for efficiency. This process is illustrated in Figure 8. Let us assume that there are random instances in a 2-dimensional space, as it is shown in Figure 8a. Then, a virtual grid is laid over the data space to create cells where there exists at least one instance, as it is shown in Figure 8b. Note that the grid is considered to be virtual because it is never materialized in practice. Each cell is represented by its coordinates in each dimension; for example, cell $(2,0)$ means that its coordinate in the first dimension is 2, and it is 0 in the second dimension. As it can be seen in Figure 8b, all non-empty cells are stored as a list; in this example, it is: $[(2,0),(3,1),(2,2),(2,3),(6,4),(4,6),(3,6),(4,7)]$. The next step is to sort the set of cells lexicographically, thus resulting in: $[(2,0),(2,2),(2,3),(3,1),(3,6),(4,6),(4,7),(6,4)]$. This sorting strategy has the time complexity of a regular sorting algorithm, which is $O(n\ log\ n)$, and it is one of the core concepts of the EGO-family of join algorithms that allow fast SJ operations.

The main advantage of this approach is that direct neighbors of any cell tend to be close to each other in the ordered set. For example, a direct neighbor of cell $(2,2)$ is any other cell that satisfies the constraint $(2\pm1,2\pm1)$, which means that cells $\{(1,1),(1,2),(1,3),(2,1),(2,3),(3,1),(3,2),(3,3)\}$ are all potential neighbors, as long as the they exist. For the join of two EGO-sorted datasets, say A and B, the naïve implementation basically splits in half each dataset. Let us assume that dataset A is split into A_1 and A_2; similarly, dataset B leads to B_1 and B_2. Then, the algorithm performs d recursive calls, where d is the number of dimensions, and computes the union of join executions in all portions of the dataset; specifically, it computes $J(A_1,B_2)\cup J(A_2,B_1)\cup J(A_1,B_1)\cup J(A_2,B_2)$ where the join operation is denoted as function $J(\ )$. The first recursive call filters all cells that do not exceed ±1 grid cell length in the first dimension; the second call filters the cells by the second dimension, and so on until the d^{th} dimension. The join operation implements several optimizations to avoid unnecessary computations.

Later, the EGO*-Join (KALASHNIKOV; PRABHAKAR, 2007) algorithm was introduced with focus on the evaluation of appropriate execution strategies depending on the number of dimensions and on their selectivity. The authors developed several heuristics to speed up the join process. Finally, Super-EGO (KALASHNIKOV, 2013) proposed data-driven dimensionality reordering heuristics that considerably avoid unnecessary computation, and also provided a parallel version of the algorithm. Despite all advancements of this approach, the fundamental concept has not changed. The solution is based on a particular criterion to sort instances, and further developments explore in depth how to do it more efficiently. One of the main advantages is that many heuristics can be employed, and no advanced data structure is required. Thus, we see the use of sort-based join methods as a promising strategy to speed up the detection of outliers.

2.3 Final Considerations

In this chapter, we introduced the fundamental concepts of two subjects; outlier detection and similarity join operation. For outlier detection, we covered the definitions, learning schemes, pipeline components, outlier types, and conventional strategies in the literature. Regarding the similarity join operation, we provided details about definitions, naive implementation, and conventional strategies in the literature that aims to overcome the performance challenges.

One interesting and crucial overlapping concept in both subjects is the proximity-based approach for solving problems. In outlier detection, virtually all algorithms rely on the Euclidean distance function for measuring the similarity among instances. Likewise, the similarity join also relies on the same function due to its useful metric property. Besides, the neighborhood notion of range and kNN is a common approach for finding similarity based on how close instances are in the dataset. Thus, these overlapping concepts allow us to investigate novel ways to detect outliers taking advantage of both subjects.

A more practical observation comes from the seminal algorithms DB-Out and aLOCI that are based on multiple executions of the range search (a.k.a. ε-neighborhood search or similarity search) operation on hypercubes. These algorithms rely on data structure's efficiency for fast outlier detection, however, the similarity join literature introduces the Super-EGO algorithm proposing an efficient way for sorting hypercubes to perform fast range searches in high-performance database systems. Therefore, considering these observations we notice there is room for performance improvement in the hypercube-based outlier detection approach which is also investigated in this work.

In the following chapters we present the related work (Chapter 3) for both subjects followed by our two novel algorithms (Chapter 4 and Chapter 5) that combines similarity join techniques with outlier detection that outperforms existing state-of-the-art algorithms in terms of performance and yet reporting high-quality results.

CHAPTER

3

RELATED WORK

3.1 Outlier Detection

The research on outlier detection started a few decades ago by the statistical community, where the fundamental concepts were developed. During all those years, several approaches have been introduced in the literature, but one that had a notorious contribution to the research community was the algorithm DB-Out (KNORR; NG, 1998). After this algorithm, a paradigm shift occurred in the community, and several alternative approaches were proposed, allowing them to achieve impressive results. The main idea of DB-Out is to interpret the detection of outliers as a spatial search problem that does not necessarily require knowledge about the underlying data distribution. This spatially oriented view allows algorithms to be employed in multidimensional spaces with greater ease compared with what was possible before, but it requires a distance function to distinguish instances in the space, so it is commonly referred to as a distance-based approach. Despite the significant benefits, DB-Out required the parameters p and D, and a distance function to be defined beforehand in order to detect outliers, where a given instance is said to be an outlier if at least a fraction p of all instances has distance greater than D from the given instance. The detection step is essentially one range query operation per instance to retrieve its neighboring instances. Since each instance is compared against all other instances, the time complexity is $O(n^2)$. Thus, the authors also introduced an approximation version with linear time complexity at the expense of large memory consumption and lower detection accuracy. The quadratic time complexity is not ideal for most applications, and neither it is the large memory consumption with low accuracy. Therefore, a number of attempts to tackle these issues have been made, as we describe in the following.

LOCI (PAPADIMITRIOU *et al.*, 2003) is another algorithm that follows a similar idea based on the range search operation. It uses a deviation factor considering the same neighborhood concept of DB-Out and also requires intensive range search operations to calculate the neighborhood density. Due to its superlinear computational complexity, the authors presented an approximation version named aLOCI that creates multiple shifted copies of the dataset and indexes these copies in Quad-trees (FINKEL; BENTLEY, 1974) using the data structures as box-

countings to estimate the neighborhood density. However, the algorithm uses three user-defined parameters, where L is the number of levels for the quad-trees, g is the number of quad-trees and *nmin* is the minimum number of instances in a leaf node to be split. The authors recommend g to be defined in the range $10 \leq g \leq 30$, which means that several trees must be created in memory, but it can lead to inefficiencies in the algorithm due to high memory consumption. In terms of accuracy, aLOCI presents inferior results compared with its non-approximated version.

Comparing all instances against each other to identify outliers is impractical because it leads to quadratic time complexity. To circumvent this situation, algorithm kNN-Out (RAMASWAMY; RASTOGI; SHIM, 2000) adopts a more efficient approach, which consists in finding the kNN of each instance using some metrics to assign an outlierness score to the instance. It can be done by using the distance to the k^{th} neighbor or the mean distance to the kNN. In this algorithm, the scores are sorted in descending order with time complexity $O(n\ log\ n)$, and only the top-N instances with the highest outlierness scores are returned. A naïve approach for this algorithm might take a time complexity $O(n^2)$; thus, the authors propose the use of some efficient data structures to search for the kNN, e.g., R-tree, but more efficient data structures that were posteriorly developed can be employed as well, like the INNA (LEE; PARK, 2005) that allows searching for the kNN in $O(log\ n)$. The authors also suggest the use of a clustering algorithm as a pre-processing tool for better data partitioning; however, this process can insert bias of the clustering algorithm into the result and potentially limit the detection of outliers.

In order to reduce the quadratic time complexity in outlier detection, LOF (BREUNIG *et al.*, 2000) further explores the idea of using kNN. It considers the ratios between the neighborhood of an instance and the neighborhoods of its neighbors. These ratios determine how dense a neighborhood is by assuming that outliers exist in regions with low density. This approach introduced the concept of a local outlier, which refers to a local evaluation considering a limited neighborhood instead of a global evaluation to determine the outlierness score of an instance. The authors also introduced a function named Local Outlier Factor (LOF) to assign such score. The original implementation is based on an X-tree (BERCHTOLD; KEIM; KRIEGEL, 1996) to perform fast kNN queries and allows the algorithm to run in $O(n\ log\ n)$. However, the authors also argue that other index-based approaches can be employed for large datasets.

Algorithm ODIN (HAUTAMÄKI; KÄRKKÄINEN; FRÄNTI, 2004) presents an alternative way to detect outliers. It performs the all-k-nearest neighbor operation and creates a graph in which each instance is represented by a vertex, and directed edges connect it with the vertices of the instance's k nearest neighbors. The outliers are therefore identified by those vertices that have the in-degree lower than a threshold D. According to the authors, the time complexity for the graph construction is $O(n\ log\ n)$, and the detection is $O(n)$ because it only checks the in-degree of each vertex.

The efficiency of these kNN-based algorithms typically depends on the data structures that they adopt. However, the approximate nearest neighbor search can also be used (ORAIR

et al., 2010). HilOut (ANGIULLI; PIZZUTI, 2002) uses the concept of Hilbert space-filling to linearize the data space and to approximate the kNN search; then, it performs multiple data scans to refine the search. The parameter k and the approximation precision h must be specified beforehand. Recently, approximate outlier detection has also been investigated in the context of ensembles to obtain diversified results (KIRNER; SCHUBERT; ZIMEK, 2017).

In recent years, the volume of data collected or generated in many scientific and commercial areas has increased considerably, and the number of dimensions has also followed this growth bringing new challenges for the outlier detection community. The issues of high dimensionality had already been noticed in the literature, but, only recently, they have become more evident to the scientific community. Indeed, the term *curse of high dimensionality* was coined decades ago in dynamic optimization problems (BELLMAN, 1961). In the context of outlier detection, the curse of high dimensionality has become evident in distance-based approaches known as concentration of distances, which assume that: *"the ratio of the variance of the length of any point vector with the length of the mean point vector converges to zero with increasing data dimensionality"*, and, as a consequence *"the proportional difference between the farthest-point distance and the closest-point distance (the relative contrast) vanishes"* (ZIMEK; SCHUBERT; KRIEGEL, 2012). This observation suggested that distance-based algorithms would become insignificant to distinguish instances in high-dimensional spaces.

Due to this reason, algorithms based on angles were proposed as an attempt to mitigate the contrast problem. The parameter-free ABOD (KRIEGEL; SCHUBERT; ZIMEK, 2008) algorithm is well known for introducing this concept in the outlier detection literature. The goal is to measure the variation of the angles of triplets of instances by performing pairwise computations of angles, assuming that outlier instances have a smaller variation of angles than regular instances. The major drawback of this approach is its cubic time complexity on the data cadinality, which is recognized by the authors. Thus, approximate solutions were proposed in order to reduce such complexity. Besides that, other techniques like FastVOA (PHAM; PAGH, 2012) propose more efficient algorithms and draw criticism about the metric Angle-Based Outlier Factor (ABOF), arguing that the algorithm ABOD should not be considered a truly angle-based method because its metric ABOF is weighted by distances.

The aforementioned algorithms inspired many others to evaluate the local density for outlier detection, aiming to improve the detection runtime and accuracy (JIN *et al.*, 2006; KRIEGEL *et al.*, 2009; KRIEGEL; KRÖGER; ZIMEK, 2009; ZHANG; HUTTER; JIN, 2009). Recently, Schubert, Zimek and Kriegel (2014) presented a generalized view on locality to unify the different (mis)interpretations of local outlier detection and improve the understanding of this type of approach in the literature. Although this paradigm is well established in the literature, other approaches attempt to solve issues that remain open. For example, some authors argue that the mere fact of adding new dimensions does not necessarily result in a difficulty in distinguishing points in space, since only the addition of irrelevant dimensions can reduce the

contrast of instances (HOULE *et al.*, 2010). In other words, this observation assumes that outliers can be identified in subspaces of lower dimensionality, i.e., subsets of the original dimensions or linear combinations thereof. Thus, the use of distance-based approaches can still be a reasonable strategy for ranking instances according to the distances to their neighbors, as long as it is performed in relevant subspaces that must be identified beforehand. Based on this observation, algorithm PINN (VRIES; CHAWLA; HOULE, 2010) exploits random projections based on nearest neighbors to reduce the space dimensionality and approximately find outliers.

Following this trend, Keller, Müller and Böhm (2012) introduced a search method named HiCS to select subspaces of high contrast for posterior density-based outlier ranking, assuming that an outlier has a low density of points in its neighborhood compared with the density of each of its closest neighbors' neighborhood. HiCS is a preprocessing tool to be applied before the use of an outlier ranking algorithm. Thus, such an approach requires a high computational cost when processing large volumes of data, especially because each of the subspaces of high contrast identified by HiCS must be analyzed in separate. There are several challenges when considering subspaces for outlier detection methods, but many interesting possibilities are open to investigation (KRIEGEL *et al.*, 2011; ZIMEK; SCHUBERT; KRIEGEL, 2012).

Although many solutions for outlier detection have been proposed over the years, they are commonly scattered in code repositories, or even different implementations of the same algorithm end up being created. As a consequence, a relevant issue occurs when the comparison between algorithms needs to be performed, and no common implementation environment is available. With that in mind, tools like ELKI (ACHTERT *et al.*, 2011) and PyOD (ZHAO; NASRULLAH; LI, 2019) have been created to facilitate fair comparison among algorithms in several datasets and to better ground the advancements in the field. Additionally, it is essential to offer graphical interface tools that provide interactivity to specialists in order to reduce coding time and accelerate the achievement of interpretable results. Following this trend, REMIX (FU *et al.*, 2017) automates the data exploration process and provides a variety of visual and interactive tools to help humans to interpret results.

Finally, although the authors of the existing algorithms have attempted to tackle many problems, new challenges continue to emerge. With the rise of smart devices and the Internet-of-Things (IoT), new approaches need to be developed to address the needs of devices with limited resources regarding both storage and processing. With that in mind, algorithm ACE (LUO; SHRIVASTAVA, 2018) proposed a locality-sensitive-hashing-based strategy for ultra-low memory devices considering privacy issues, while Yu, Wang and Shami (2017) proposed an algorithm for the analysis of sequential events in real time based on the recursive Principal Component Analysis (PCA). This new trend may be able to take advantage of all accumulated knowledge of the current literature to push the research on outlier detection towards new solutions.

3.2 Similarity Join in Data Mining

In recent years, the use of data mining has become crucial for most businesses. It allows teams to uncover hidden patterns or ensure the data quality in large datasets so to make more well-grounded decisions. In this setting, similarity self-join algorithms are often applied to support data mining in several tasks aimed at delivering valuable results efficiently.

Whenever heterogeneous sources of data are merged together into a single dataset, non-identical duplicate entities may emerge. In this case, the entities are referred to by non-unique descriptions, which cause ambiguity and reduce the quality of the data. For example, in a given dataset, the person "John Snow" (entity) might have the name (description) misspelled or abbreviated as "J. Snow". In both cases, the same entity is differently referred to. Two related processes aim to tackle this problem: Record Linkage (RL) and Entity Resolution (ER). The former consists of determining whether two records of interest are the same or not, and the latter consists of ensuring that references in a dataset point to the correct entities. Since both processes are tightly related, the existing RL approaches can be adapted for ER (KALASHNIKOV; MEHROTRA, 2006). A common approach for both RL and ER is to apply similarity self-join by computing pairs of references using a similarity function and comparing the similarity score with the threshold ε (KALASHNIKOV; MEHROTRA, 2006; CHEN; KALASHNIKOV; MEHROTRA, 2009; WANG *et al.*, 2011; ADHAV; KUMAR, 2015; GIANNAKOPOULOU *et al.*, 2017). When the similarity score is higher than the threshold, the corresponding pair is flagged as being coreferent. This approach provides flexibility to exploit textual similarity functions (WANDELT *et al.*, 2014) or the interdependence between references (CULOTTA; MCCALLUM, 2005; DONG; HALEVY; MADHAVAN, 2005).

Clustering is another data mining task that may take advantage of the similarity self-join operation. Seminal clustering algorithms such as DBSCAN (ESTER *et al.*, 1996) inspired many researchers to develop clustering techniques that heavily rely on successive similarity search operations. With that in mind, Böhm *et al.* (2000) proposed a generic schema to transform these algorithms into an adapted representation that uses the similarity self-join operation. This schema produced significant improvements compared with the original algorithms. Also, the authors mention other data mining tasks as examples were their ideas could be successfully applied, including outlier detection.

The similarity self-join operation has been successfully applied in several data mining tasks, and the use of such operation for outlier detection has been mentioned over the years (BÖHM *et al.*, 2000; BÖHM *et al.*, 2001; XIA *et al.*, 2004; KALASHNIKOV; PRABHAKAR, 2007; BRYAN; EBERHARDT; FALOUTSOS, 2008; LU *et al.*, 2012; LUO *et al.*, 2012). However, to the best of our knowledge, the details regarding the implementation of this idea to detect outliers and experiments in real-world datasets remain missing in the literature. In order to fill this gap, we aim at taking advantage of the efficiency of the similarity self-join operation to demonstrate that it can be a compelling alternative in the outlier detection task.

3.3 Final Considerations

In this chapter, we described different algorithms that detect outliers and discussed some popular tools that implement several state-of-the-art algorithms aimed at benchmark evaluation. In order to summarize the discussion, Table 1 highlights the main aspects of relevant algorithms including the time complexity and other important characteristics.

Algorithm	Time Complexity	Approach	# Parameters	Deterministic	Data Structure
aLOCI	$O(ndgL)$	Density	3	No	Quad-Tree
LOF	$O(nd\ log\ n)$	Density	1	Yes	X-Tree
ODIN	$O(nd\ log\ n)$	Distance	2	Yes	Graph
kNN-Out	$O(nd\ log\ n)$	Distance	1	Yes	R*-Tree
DB-Out	$O(n^2d)$	Distance	1	Yes	-
HilOut	$O(n^2d^2)$	Distance	2	Yes	-
ABOD	$O(n^3d)$	Angle[1]	0	Yes	-

Table 1 – Summary of state-of-the-art algorithms in outlier detection. The number of instances n, the number of dimensions d, the number of trees g and the number of tree levels L denote the variables that affect the time complexity of each algorithm.

The algorithms are sorted in ascending order of time complexity. The first one is aLOCI, which has a $O(n)$ time complexity due to its approximate way of detecting outliers. It assumes that there is a gaussian data variation in the neighborhood of each instance. Even so, it is essential to note that the input parameters L and g, that is, the tree depth and the number of trees, respectively, can also affect the runtime. For instance, the authors recommend the creation of 10 to 30 independent tree structures to index the data instances, i.e., $10 \leq g \leq 30$. These parameters can increase the algorithm runtime dramatically, thus making aLOCI impractical for large datasets. As each tree is generated by randomly shifting instances in the space, this causes the algorithm to return different results at each execution and to depend on randomness for accurate results. Besides, the fact that there are 3 user-defined parameters makes the algorithm more sensitive, especially with regard to parameter g that must be carefully defined. Moreover, for each neighborhood search, the algorithm needs to evaluate 2^d cells in the worst-case scenario.

The kNN-based algorithms that use indexing data structures, that is, LOF, ODIN and kNN-Out, show a worse time complexity compared with aLOCI. However, except for ODIN, they require only the parameter k to detect outliers. Here, the main challenge is to mitigate the indexing costs for large datasets. For example, ODIN requires a kNN-graph to be constructed as a preprocessing step so that later the algorithm evaluates the in-degree of each instance/vertex to find the outliers. Moreover, to test if a new instance is an outlier or not, the entire graph needs to be reconstructed, which is not desired for real-world applications that prioritize performance.

HilOut is also kNN-based and relies on approximate results according to the Hilbert space-filling curve, which can be efficiently computed. Another strategy that helps the algorithm

[1] It also considers distances as weights in the process.

to reduce the runtime is to report only the instances that show a high outlierness score, i.e., the top-N outliers, instead of reporting the outlierness score for every instance. In scenarios where the outlierness score for every instance is required, the algorithm may have a quadratic time complexity with respect to the number of instances and dimensions. Finally, DB-Out and ABOD do not use any indexing structure and show quadratic and cubic time complexities, respectively.

As a concluding remark, let us highlight the facts that: (i) outlier detection is commonly posed as a similarity search problem, and; (ii) the use of indexing data structures to reduce the time complexity is a popular strategy among the state-of-the-art algorithms. However, the index-based approach may struggle to offer scalable and distributed solutions to process large volumes of data. As it was previously mentioned in this work, the similarity self-join operation is designed to efficiently handle the costs of intense search operations in large datasets. Here, it is important to note that the index-based approach is solely one alternative among others for the join operation; see the previous Section 2.2 from Chapter 2 for details. With that in mind, in the following chapter we discuss the link between outlier detection and similarity self-join, and introduce the first main contribution of this MSc work: one novel algorithm to efficiently and accurately detect outliers by following a join-based approach.

CHAPTER

4

OUTLIER DETECTION WITH SIMILARITY SELF-JOIN

Similarity self-join is a fundamental operation in database systems that is used in many data mining tasks due to its high performance. On the other hand, outlier detection is an important data mining task that strives to achieve low runtime in large datasets. Here, we detail the missing link between both concepts and introduce a novel similarity-self-join-based algorithm for outlier detection that can reduce the detection runtime considerably without compromising the accuracy of results. Specifically, this chapter is organized as follows: in Section 4.1, we present the link between outlier detection and similarity self-join by demonstrating their relationship from theoretical and practical standpoints; Section 4.2 formalizes our targeted outlier problem; Section 4.3 introduces in detail a framework that takes advantage of existing similarity self-join algorithms to spot outliers using a threshold-based approach; in Section 4.4, we describe the setup of a comprehensive experimental evaluation that was performed to validate our proposals; Section 4.5 reports and discusses the results obtained when comparing our proposed algorithm with the state of the art; finally, the concluding remarks of the chapter are presented in Section 4.6.

4.1 Outlier Detection meets Similarity Self-Join

In the previous chapters, we have introduced the fundamentals of outlier detection and similarity search, which notably share concepts among each other. In the outlier-related literature, algorithms that are based on similarity search are often named as neighborhood-based (CHANDOLA; BANERJEE; KUMAR, 2009) or proximity-based algorithms (AGGARWAL, 2017), and, depending on the aspect taken into account, they may also be named as density-based algorithms (CHANDOLA; BANERJEE; KUMAR, 2009; AGGARWAL, 2017). In this section, we describe our understanding of the shared concepts and exemplify from a theoretical standpoint the reason why similarity self-join can improve the efficiency of outlier detection.

For clarification, we understand as neighbor, aka nearest, closest or proximal, any instance that lies in the neighborhood of a query instance q. The neighborhood or vicinity is the region where all neighbors of q lie in the feature space. This region can be understood in many ways, but

it is usually defined in terms of the radius from the query instance. For example, the neighborhood radius in a range search is defined by the parameter ε, while in the kNN search it is given by the distance to the k^{th} neighbor. Therefore, the neighborhood radius may be different for distinct query instances when using the kNN search. On the other hand, in the range search the radius is always the same for any query instance.

A similar instance is any instance that resembles an instance of interest q. Note that identical instances can or cannot be taken as similar depending on the application. However, resemblance is a vague concept that can be defined in many ways. In our formulation, we consider to be similar any instance that lies in the neighborhood of q; that is, if two instances are close enough so that they share the same neighborhood in the feature space, they are similar to each other. Thus, the definition of neighbor and similar share the same semantics, and we use both terms interchangeably.

Many similarity search tasks rely on the properties of the metrics to find neighboring instances efficiently, and the outlier detection task often depends on metric distance functions, e.g., the Euclidean distance (AGGARWAL, 2017, p. 117). With that in mind, we shall argue that these outlier detection approaches do not take full advantage of the metric properties because they do not use the symmetry property to prune the space search. For example, let X denote a dataset and $x, y \in X$ be neighboring instances that are within a range of radius ε. In this setting, the outlier detectors would require one similarity search operation to discover that x (as query center) is a neighbor of y, and another one to detect that y (as query center) is a neighbor of x. On the other hand, the similarity self-join would perform a single operation to find out that x is a neighbor of y and vice versa, so it does not require a second operation. This pruning is possible for two reasons: (i) under the self-join formulation it is known beforehand that the goal is to find all neighboring instances of X, and; (ii) the symmetry property is satisfied. Thus, we see this theoretical observation as one major contributor to support the hypothesis of our work.

As it was discussed in the previous Chapter 2, successive similarity search operations may be replaced by one single similarity self-join operation – see Section 2.2, which aims to take advantage of metric properties to prune the search space and produce the same result more efficiently (DOHNAL; GENNARO; ZEZULA, 2003; KALASHNIKOV, 2013; FREDRIKSSON; BRAITHWAITE, 2015). Meanwhile, the neighborhood-based outlier detection literature heavily depends on the range and the kNN search operations – see Section 2.1.3, just like it happens with the similarity search literature. Since both tasks rely on the same fundamental search operations and differ in their goal, we believe that a better comprehension of the differences would clarify how to use similarity self-join to speed up outlier detection.

The major difference between these tasks is that the similarity self-join aims at finding all pairs of similar instances, that is, all pairs of neighbors. On the other hand, neighborhood-based outlier detection spots instances that do not appear in any of these pairs or appear only in a few of them, i.e., instances with no neighbor or few neighbors, which could be outliers due to their lack

of resemblance to other instances. Note that in both tasks it is required to find the neighbors of each instance. It means that a naïve neighborhood-based outlier detector would use all instances of the dataset as query centers for similarity search operations, which is exactly the same as one similarity self-join operation. Thus, the main challenge to develop a join-based outlier detector is to adapt the similarity self-join algorithm to identify only part of the pairs of neighbors, and not all pairs, in such a way that it stops looking for additional neighbors of instances that already have many neighbors identified. By this observation, it is expected for the adapted solution to be even faster than the original join operation because not every pair of similar instances must be identified. Note that most similarity join algorithms aim specifically at improving the operation efficiency, since the query result is always the same for exact algorithms, and still, in the context of outliers, this efficiency can be improved even further by considering the narrower goal of outlier detection. In this sense, we believe that many neighborhood-based outlier detectors may benefit from the similarity self-join operation to reduce their runtime. In the following sections, we provide the problem statement, introduce our proposal and show how to adapt any similarity self-join algorithm to spot outliers efficiently while reporting highly accurate results.

4.2 Problem Statement

Our problem follows the well-accepted (SCHUBERT; ZIMEK; KRIEGEL, 2014; CAMPOS *et al.*, 2016; GOLDSTEIN; UCHIDA, 2016) assumption that instances with few neighbors are very likely outlying instances, and their closest instances may also be outliers. We consider that the dataset to be analyzed is represented in a d-dimensional space, and that it includes m instances, which are distinguished among each other using a distance function. In this setting, an instance is *flagged* as an outlier when the number of instances in its neighborhood is smaller than or equal to a predefined threshold τ. This concept is formalized in Definitions 4 and 5.

Definition 4 (Range Search)**.** The range search operation is expressed by a function $rng(X,q,\varepsilon)$ that returns all instances of a dataset X whose distances to a given query center instance q are within a predefined radius ε, according to a distance metric function f. Formally, it is given by:

$$rng(X,q,\varepsilon) = \{x : x \in X \ \wedge \ f(x,q) \leq \varepsilon\}$$

Definition 5 (Outlier Flagging)**.** An instance q in a dataset X is *flagged* as an outlier when the number of instances in its neighborhood of radius ε is smaller than or equal to a predefined threshold τ, where $1 \leq \tau \leq m$. Formally, it is given by:

$$Flag(X,q,\varepsilon) = \begin{cases} True, & \text{if } |rng(X,q,\varepsilon)| \leq \tau \\ False, & \text{otherwise} \end{cases}$$

These definitions describe the exact solution for the outlier detection problem that we investigate. The following section provides the details of our proposed algorithm.

4.3 The ODSSJ Algorithm

This section provides the details of our proposed **O**utlier **D**etection algorithm based on **S**imilarity **S**elf-**J**oin – the **ODSSJ** algorithm. A generic pseudocode that illustrates our approach is presented in Algorithm 1. Note that we use functions in the pseudocode as placeholders for specific algorithms. That is, function *SSJ* is a placeholder for any similarity self-join algorithm; similarly, function *OD* is a placeholder for the criterion to be used to identify outliers. The functions' parameters were omitted because they depend on the specific algorithms to be selected for the implementation, and, in practice, their values must be defined by specialist users in the application. Note that our pseudocode focuses on an in-memory and single-thread implementation, but the same idea could be used in a building-block fashion for other settings.

Algorithm 1 is a generic approach that uses similarity self-join for outlier detection. In our notation, we assume that the input dataset $X = [x_1,\ x_2,\ \ldots\ x_m]$ has m instances, where each instance $x_p = [x_{p,1},\ x_{p,2},\ \ldots\ x_{p,d}]$ is itself a d-dimensional data point. One array of boolean values $O = [o_1,\ o_2,\ \ldots\ o_m]$ is the output; each element o_p indicates whether the instance x_p is an outlier or not, so o_p is the outlier flag. Our join-based proposal for outlier detection is divided into two steps. At first, function *SSJ* is responsible for finding the neighbors of each instance $x_p \in X$ with respect to Definition 4. Since we are interested in the numbers of neighbors, and not in the neighbors themselves, there is no need for *SSJ* to return pairs of similar instances as usual. Instead, we consider that function *SSJ* returns an array $C = [c_1,\ c_2,\ \ldots\ c_m]$, where each element c_p is the number of neighbors of instance x_p. Then, function *OD* evaluates the number of neighbors c_p of each instance x_p and reports the corresponding boolean flag o_p as in Definition 5.

Algorithm 1 – `ODSSJ( )`

Input: Dataset X.
Output: Array O of outlier flags.

1: $C \leftarrow SSJ(X)$;
2: **for** each c_p in C **do**
3: $\quad o_p \leftarrow OD(c_p)$;
4: $\quad$ Store o_p as the value for the p^{th} position of array O;
5: **end for**
6: **return** O;

To implement function *OD*, we follow Definition 5 and compute outlier flags according to the number of neighbors of each instance. As in the definition, we consider the existence of a threshold τ that sets the maximum number of neighbors accepted for outliers. When the number of neighbors c_p is smaller than or equal to τ, the corresponding instance x_p is flagged as an outlier, that is, its label is *True*; otherwise, the label is *False*. Formally, it is given by:

$$OD(c_p) = \begin{cases} True, & \text{if } c_p \leq \tau \\ False, & \text{otherwise} \end{cases} \tag{4.1}$$

A naïve implementation for function *SSJ* would perform multiple range search operations, one per instance $x_p \in X$. It would compare all instances against each other, and, any two instances with distance smaller than or equal to the threshold ε would be considered neighbors or similar to each other. Such implementation would lead to a quadratic time complexity, which is unacceptable for most applications. Thus, we propose to implement function *SSJ* by carefully adapting the Super-EGO similarity join algorithm for fast outlier detection. As it was described in Section 2.2.3 from Chapter 2, Super-EGO employs strategies to mitigate the quadratic time complexity by sorting the instances in a way that similar instances are stored close to each other, and, recursively partitioning the set of instances into subsets until the subsets have a minimum cardinality, so that they are efficiently joined using a naïve join function named *SimpleJoin*. The minimum cardinality number is the threshold parameter t defined by the specialist beforehand.

Provided that function *OD* uses counts of neighbors instead of the neighbors themselves, and it flags as inliers all instances with more than τ neighbors, no matter how many more neighbors they have, one can speed-up Super-EGO for outlier detection. Specifically, we propose to incorporate two optimizations into its *SimpleJoin* function. The first one is to only store (in-memory) the number of neighbors for each instance, as opposed to the original algorithm that stores all the neighbors themselves. In the second optimization[1], the threshold τ is incorporated into function *SimpleJoin* to avoid unnecessary comparisons; that is, whenever it is known that an instance has more than τ neighbors, the join process stops because to look for additional neighbors would not change the decision of the outlier detector. These optimizations allow the adapted Super-EGO to require considerably smaller amounts of main memory, compared with the requirements of the original algorithm, and they also avoid the identification and output of huge sets of pairs of similar instances that commonly occur in dense regions of the feature space.

In this way, our proposed algorithm ODSSJ follows the general idea of Algorithm 1, considering that function *OD* is implemented as in Equation 4.1 and that function *SSJ* uses an adapted version of the Super-EGO algorithm, which incorporates the aforementioned optimizations. Note that our proposal could be extended for distributed and multi-thread settings because the neighborhood counting only requires readings from dataset X and the evaluation only requires a predefined threshold. In the following sections, we provide the experimental setup, report and discuss the results, and present the final remarks of this chapter.

[1] Note that this optimization is described by Aggarwal (2017) as counting neighborhoods in histogram-based techniques.

4.4 Experimental Setup

This section presents the experimental setup. Our proposed algorithm ODSSJ was implemented in plain Java. We compared it with 7 state-of-the-art algorithms, namely *k*NN-Out, DB-Out, LOF, ODIN, HilOut, aLOCI and ABOD, which are also implemented 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 ELKI's default in-memory hashtable that is expected to be faster than the other alternatives[2]. Also, we used the Euclidean distance for all applicable algorithms. To ensure a fair comparison among the competitors, we performed an exhaustive parameter search for each algorithm and dataset pair to identify the result with the highest accuracy, which is reported. Appendices A and C present the best parameter configuration identified for each algorithm and dataset. **Observation:** for the purpose of reproducibility, all codes, detailed results, parameter values tested and datasets used in this chapter are freely available for download online[3].

Table 2 has a summary of the 12 real-world datasets that we studied. They were all obtained from the UCI Machine Learning Repository (DHEERU; TANISKIDOU, 2017), being selected for this evaluation due to their constant use in the literature on benchmarks for outlier detection (EMMOTT *et al.*, 2015; CAMPOS *et al.*, 2016). All datasets are labeled indicating whether each instance is an outlier or not. Obviously, the labels are only used for accuracy evaluation as we focus on unsupervised learning. Moreover, the preprocessing used for each dataset is described in Rayana (2016). For our proposed algorithm, we also performed a normalization in all datasets to set an interval for parameter ε and to facilitate the parameter search.

Dataset	# Instances	# Dimensions	# Outliers	% Outliers
parkinson	50	22	2	4.00
hepatitis	70	20	3	4.29
glass	214	9	9	4.21
ecoli	336	7	9	2.68
ionosphere	351	33	126	35.90
breastw	683	9	239	34.99
pima	768	8	268	34.90
thyroid	3,772	22	93	2.47
satimage2	5,803	36	71	1.22
mammography	11,183	6	260	2.32
shuttle	49,097	9	3,511	7.15
http	567,479	3	2,211	0.39

Table 2 – Summary of datasets

The experiments were performed on one machine that has a processor Intel© Core™ i7-2600S with 4 cores at 2.8GHz and 8GB of RAM, running GNU/Linux Ubuntu Xenial x86-64. To evaluate accuracy, we used the ROC AUC (AUC) metric due to its popularity in the unsupervised

[2] <https://elki-project.github.io/releases/current/doc/de/lmu/ifi/dbs/elki/database/StaticArrayDatabase.html>
[3] <https://github.com/eug/odssj>

outlier detection literature (CAMPOS *et al.*, 2016). The AUC score ranges from 0 to 1, where a perfect detection has value 1, and a random detection has value ~ 0.5. For the evaluation of efficiency, the runtime in seconds of each execution was aggregated by average and standard deviation after 10 independent executions, always considering the parameter configuration that leads to the highest AUC value for each algorithm and dataset pair. Note that the loading time and the index-building time (whenever it is applicable) of each algorithm were ignored.

4.5 Results and Discussion

This section reports and discusses the results of our experimental evaluation under the setup of Section 4.4. We aimed at answering the following questions:

Q1 Compared with 7 of the recent and related works, how accurate is our proposed ODSSJ ?

Q2 How efficient are the techniques studied?

Q3 How different is the behaviour of ODSSJ compared with that of the other techniques, that is, are the differences statistically significant regarding runtime and accuracy measurements?

Here, it is important to note that we tested every one of the techniques studied in every dataset, but ABOD, aLOCI and HilOut failed to run in the largest dataset *http* because they exceeded the main memory capacity in every possibility of the exhaustive parameter search that we performed. ABOD also presented the same behaviour for the second largest dataset *shuttle*. In our opinion, none of the datasets studied is large enough to justify a need for more than 8GB of RAM. Therefore, we do not report results regarding these particular cases.

4.5.1 Evaluation of Effectiveness

This section investigates Question **Q1**: *"Compared with 7 of the recent and related works, how accurate is our proposed ODSSJ?"*. To make it possible, Table 3 provides the AUC score values obtained from testing the algorithms in each dataset; remember that we report the highest score obtained with exhaustive parameter search from each algorithm and dataset pair. The average of these highest scores for each algorithm is also shown in Table 3 to highlight the general trends. Note that the average values only consider the reported measurements, so they disregard dataset *http* for algorithms HilOut, aLOCI and ABOD, besides dataset *shuttle* for ABOD. With that being said, it can be seen in Table 3 that our proposed ODSSJ provided the highest average value among the algorithms studied, and HilOut reports similar results on average, but unable to process the *http* dataset.

Regarding the results for each dataset in particular, our ODSSJ delivered the highest scores for 4 out of the 12 datasets studied, while ABOD obtained the best results for 3 datasets each, HilOut and *k*NN-Out was the most accurate algorithm for 2 datasets, and LOF was victorious in only 1 dataset. Note that all algorithms reported scores close to 0.5 in at least one dataset, which is the score expected for a random detection; that is, no algorithm consistently delivered accurate results regardless of the dataset. Many of the near-random results were obtained with dataset *hepatitis*, so it can be considered as the most challenging one. In this particular dataset, the highest score was 0.6567, which was obtained by algorithm HilOut, while our ODSSJ obtained a very similar score of 0.6493. To summarize, the aforementioned results indicate that the new algorithm ODSSJ is highly accurate; compared with 7 algorithms from the state of the art, it was the most accurate on average considering 12 well-known benchmark datasets.

4.5.2 Evaluation of Efficiency

This section investigates Question **Q2**: *"How efficient are the techniques studied?"*. To make it possible, Table 4 provides the average runtime in seconds and the corresponding standard deviation for 10 independent executions of the studied algorithms in each dataset. The overall average runtime of each algorithm, i.e., the average of the averages, is also reported to highlight the general trends. Remember that we were unable to run algorithms HilOut, aLOCI and ABOD in dataset *http*, as well as ABOD in dataset *shuttle* due to main memory exceed. In spite of that fact, we still report the overall average values for these particular algorithms, but it must be remembered that these values disregard the largest datasets, so they tend to be considerably smaller than those that would have been obtained if all datasets were taken into account.

With that being said, note that there is a substantial difference in the overall average runtime among the algorithms studied; in particular, the values for algorithms HilOut, aLOCI and our proposed ODSSJ are at least 2 orders of magnitude smaller than those of the other algorithms. Since the results of HilOut and aLOCI disregard the largest dataset *http*, we argue that our ODSSJ is the most efficient algorithm among the ones studied. In fact, we performed an additional experiment to estimate the effects of the inclusion of dataset *http* in the overall average values of HilOut and aLOCI. Specifically, we created and processed smaller versions of *http* with random samples of increasing sizes up to the full dataset, i.e., 1%, 10%, 15%, 25%, 50%, 75% and 100%. Note that we randomly sampled proportional percentages of inlier and outlier instances to keep the original data characteristics. In this particular experiment, aLOCI and HilOut exceeded the main memory capacity after the 15% and the 75% sample sizes, respectively, and the average runtime of 10 independent executions with the largest sample size that they managed to analyze is respectively ~ 400 and $\sim 1,750$ seconds. More details about this experiment are given latter, in Section 5.4.3 of Chapter 5. Provided that the aforementioned runtime measurements are substantially larger than the ~ 27 seconds required by our ODSSJ to

process the whole dataset *http*, we conclude that the overall average runtime of HilOut and aLOCI would be considerably larger than the values reported in Table 4 if all datasets were taken into account.

Regarding the results for each dataset in particular, our ODSSJ delivered the lowest runtime in all cases, except for dataset *shuttle* in which HilOut obtained the best result with 1.08 ± 0.05 seconds versus the 3.11 ± 0.07 seconds required by our algorithm. Note that in Table 4 the datasets are shown in ascending order according to the number of instances. For datasets with less than 5K instances, i.e., before *satimage2*, the results are similar for all algorithms, including ODSSJ . The only exception is ABOD, for which we observe a substantial runtime increase in datasets with more than 1K instances, i.e., after *pima*, until it can no longer process larger datasets under the same conditions as the other algorithms. One can justify such behavior due to ABOD's well-known cubic time complexity and the lack of strategies to optimize its search operation. In the dataset with the largest number of instances, that is, *http*, the algorithms usually take *thousands of seconds* to detect outliers. The only exception is our proposed algorithm ODSSJ , which detects outliers in less than *thirty seconds*. Note that our algorithm aims to find neighboring instances for all instances, and yet it is up to 3 orders of magnitude faster than the existing approaches. This is due to the efficiency of the self-join algorithm combined with our optimization strategies described in Section 4.3. Here, it is important to note that the optimization strategies that we propose are one of the major contributors to reduce the runtime. In order to corroborate this fact, we performed an additional experiment using the largest dataset *http*. Without our optimizations, the runtime of ODSSJ was nearly 100 times larger than the values reported, and yet our algorithm's runtime would remain similar to those of the state-of-the-art algorithms. To summarize, the aforementioned results indicate that the new algorithm ODSSJ is very efficient; compared with 7 algorithms from the state of the art, it was up to 3 orders of magnitude faster than the fastest related works considering 12 well-known benchmark datasets.

4.5.3 Statistical Evaluation

This section investigates Question **Q3**: *"How different is the behaviour of ODSSJ compared with that of the other techniques, that is, are the differences statistically significant regarding runtime and accuracy measurements?"*. To make it possible, we conducted a pairwise comparison between our proposed algorithm and each one of the other algorithms to determine whether or not the differences in accuracy from Table 3 and in runtime from Table 4 are statistically significant. We used a two-tailed t-test with a significance level of 0.05 for this task. For runtime and accuracy we consider the results of 10 independent executions in each dataset and algorithm pair, although the accuracy results remains the same in all 10 executions.

Table 5 summarizes our findings for accuracy in the left column and runtime in the right column; it reports the count of datasets where our ODSSJ wins/loses/ties compared with the related work indicated in each row. The largest counts are highlighted in bold. Note that we

count as *win* for ODSSJ when the related algorithm fails to detect outliers in a given datasets.

As it can be seen, the results indicate that our proposed algorithm is statistically superior to all other algorithms in terms of runtime and it is also very competitive with regard to accuracy. HilOut and *k*NN-Out perform better than our ODSSJ in accuracy, with $5/7/0$ wins/loses/ties for ODSSJ versus HilOut and $5/7/0$ wins/loses/ties for ODSSJ versus *k*NN-Out, but note that there are actually minor differences in most of the accuracy results shown in Table 3 for these three algorithms, while our ODSSJ outperforms the 5 remaining related works in accuracy. On the other hand, our ODSSJ is able to process all datasets and in terms of runtime it is statistically superior to all state-of-the-art algorithms, being up to 3 orders of magnitude faster than the fastest related works considering 12 well-known benchmark datasets. Therefore, the statistical evaluation corroborates the conclusions of the previous two sections, i.e., that our ODSSJ is considerably more efficient than the existing techniques, and still it is as accurate as them.

	ODSSJ	
	AUC score (wins/loses/ties)	Runtime (wins/loses/ties)
*k*NN-Out	5/**7**/0	**12**/0/0
DB-Out	**8**/4/0	**12**/0/0
LOF	**8**/4/0	**12**/0/0
ODIN	**8**/4/0	**12**/0/0
HilOut	5/**7**/0	**11**/1/0
aLOCI	**10**/2/0	**12**/0/0
ABOD	**10**/2/0	**12**/0/0

Table 5 – Results of a pairwise comparison between ODSSJ and each one of the other algorithms studied. We report the count of datasets where ODSSJ wins/loses/ties compared with the related work indicated in each row. A two-tailed t-test with a significance level of 0.05 was used for this task. The largest counts are highlighted in bold.

4.6 Conclusion

This chapter introduced the first main contribution of the MSc work: one novel algorithm named ODSSJ that efficiently and accurately detects outliers by following a join-based approach. Our specific contributions are listed in the following:

C1 Outlier detection meets similarity self-join: we highlighted the fact that the similarity self-join operation has a strong conceptual correlation with outlier detection. It turns out to be relevant because outlier-related researches are generally focused on the accuracy aspect of the problem, while the join community has been focusing on efficiency issues for decades. With that in mind, we demonstrated both in theory and in practice how to take advantage of the knowledge acquired by the join community to detect outliers faster;

C2 Efficiency, effectiveness and versatility: we presented a novel framework for fast and accurate outlier detection by taking advantage of the fundamental database operation similarity self-join. In theory, our framework is general enough to allow distributed and multi-thread implementations due to its independence in the computation. It can also be implemented as a database management system's physical operator due the strong correlation with the similarity self-join;

C3 Experimental evaluation: we reported experimental results using 12 real-world datasets with up to $\sim 500k$ instances that are largely used in benchmarks by the outlier detection community. These results indicate that our proposed ODSSJ is hundreds or even *thousands* of times faster than 7 state-of-the-art algorithms, especially when considering large datasets, and it still obtains similar accuracy of results.

In the following chapter we revisit the hypercube-based notion for outlier detection and combine it with a sorting strategy that is commonly employed in similarity join algorithms. As a result, we present another outlier detection algorithm named HySortOD that scales to even larger volumes of data, and still provides highly accurate results.

Dataset	ODSSJ	kNN-Out	DB-Out	LOF	ODIN	*HilOut	*aLOCI	*ABOD
parkinson	**0.9896**	0.8958	0.9271	0.7917	0.7188	0.9688	0.5000	0.8438
hepatitis	0.6493	0.5473	0.6219	0.5124	0.6517	**0.6567**	0.5000	0.4328
glass	**0.8488**	0.7824	0.7897	0.6043	0.6724	0.8423	0.6382	0.7572
ecoli	**0.9557**	0.8899	0.6096	0.6070	0.9018	0.8964	0.7851	0.9059
ionosphere	0.9108	0.9303	0.8910	0.9067	0.8526	**0.9467**	0.5000	0.8961
breastw	0.9632	0.9782	0.9785	0.5989	0.8166	0.9730	0.9430	**0.9960**
pima	**0.7973**	0.6864	0.6790	0.6240	0.6692	0.7185	0.5608	0.6688
thyroid	0.8182	**0.9793**	0.9624	0.8913	0.6896	0.9779	0.9110	0.9508
satimage2	0.9904	0.9924	0.9919	0.9875	0.9931	0.9852	0.4152	**0.9972**
mammography	0.8071	0.8597	0.8772	0.8285	0.8551	0.8405	0.7572	**0.8844**
shuttle	0.9054	**0.9953**	0.6688	0.9827	0.9883	0.9335	0.9638	-
http	0.9892	0.9987	0.9891	**0.9995**	0.9746	-	-	-
Average	**0.8969 ± 0.09**	0.8779 ± 0.19	0.8321 ± 0.20	0.7778 ± 0.24	0.8153 ± 0.18	*0.8854 ± 0.10	*0.6794 ± 0.19	*0.8333 ± 0.17

* The average values only consider the reported measurements.

Table 3 – The AUC scores obtained from ODSSJ and 7 related works on 12 benchmark datasets. The largest values are highlighted in bold.

Dataset	ODSSJ Avg±Std	*k*NN-Out Avg±Std	DB-Out Avg±Std	LOF Avg±Std	ODIN Avg±Std	*HilOut Avg±Std	*aLOCI Avg±Std	*ABOD Avg±Std
parkinson	**0.01 ± 0.01**	0.02 ± 0.01	0.02 ± 0.01	0.03 ± 0.01	0.02 ± 0.01	0.03 ± 0.01	0.02 ± 0.01	0.06 ± 0.01
hepatitis	**0.01 ± 0.01**	0.02 ± 0.01	0.02 ± 0.01	0.04 ± 0.01	0.02 ± 0.01	0.03 ± 0.01	0.02 ± 0.01	0.06 ± 0.01
glass	**0.01 ± 0.01**	0.03 ± 0.01	0.02 ± 0.01	0.04 ± 0.01	0.03 ± 0.01	0.05 ± 0.01	0.03 ± 0.01	0.25 ± 0.03
ecoli	**0.01 ± 0.01**	0.04 ± 0.01	0.05 ± 0.01	0.05 ± 0.01	0.06 ± 0.01	0.10 ± 0.01	0.02 ± 0.01	0.54 ± 0.02
ionosphere	**0.02 ± 0.01**	0.05 ± 0.01	0.04 ± 0.01	0.06 ± 0.02	0.05 ± 0.01	0.09 ± 0.01	0.03 ± 0.01	0.71 ± 0.01
breastw	**0.01 ± 0.01**	0.06 ± 0.01	0.07 ± 0.01	0.08 ± 0.01	0.06 ± 0.01	0.10 ± 0.01	0.07 ± 0.01	1.80 ± 0.13
pima	**0.02 ± 0.01**	0.11 ± 0.02	0.07 ± 0.01	0.19 ± 0.03	0.12 ± 0.02	0.06 ± 0.01	0.03 ± 0.14	4.40 ± 0.19
thyroid	**0.04 ± 0.01**	0.27 ± 0.01	0.32 ± 0.01	0.46 ± 0.03	0.28 ± 0.01	0.22 ± 0.01	0.16 ± 0.01	$1x10^3 \pm 29.29$
satimage2	**0.14 ± 0.01**	1.42 ± 0.05	2.61 ± 0.07	1.93 ± 0.10	2.70 ± 0.04	0.61 ± 0.01	0.49 ± 0.02	$3x10^3 \pm 1x10^2$
mammography	**0.17 ± 0.02**	7.76 ± 1.09	7.70 ± 0.72	2.98 ± 0.26	9.59 ± 0.76	0.59 ± 0.07	6.26 ± 0.10	$3x10^4 \pm 1x10^4$
shuttle	3.11 ± 0.07	$1x10^2 \pm 0.78$	$2x10^2 \pm 3.87$	$1x10^2 \pm 9.05$	$2x10^2 \pm 2.31$	**1.08 ± 0.05**	8.59 ± 0.13	-
http	**27.14 ± 0.85**	$5x10^3 \pm 6x10^2$	$2x10^4 \pm 7x10^2$	$2x10^3 \pm 63.90$	$5x10^3 \pm 1x10^2$	-	-	-
Average	2.55 ± 7.45	$4x10^2 \pm 1x10^3$	$2x10^3 \pm 6x10^3$	$1x10^2 \pm 5x10^2$	$3x10^2 \pm 1x10^3$	***0.27 ± 0.34**	*1.43 ± 3.01	$*4x10^3 \pm 1x10^4$

* The average values only consider the reported measurements.

Table 4 – The runtime in seconds obtained from ODSSJ and 7 related works on 12 benchmark datasets. The smallest values are highlighted in bold.

CHAPTER

5

OUTLIER DETECTION WITH SORTED HYPERCUBES

The use of hypercubes in outlier detection has its origins in the seminal paper of Knorr and Ng (1998) where the DB-Out algorithm was introduced. The hypercubes are used in an optimized version of DB-Out as a way to reduce the number of entities to process by grouping close instances together into the same hypercube; it speeds the neighborhood counting up by using a naïve search strategy. Another example is aLOCI (PAPADIMITRIOU *et al.*, 2003) that uses hypercubes indexed in a quad-tree-like structure as a way to approximately identify the neighborhood density of an instance. In both algorithms, the use of hypercubes was introduced as an approximation strategy to mitigate the costs of the detection of outliers. Yet, the major challenge for these algorithms was to reduce the elevated runtime with focus on relatively small datasets. In other words, the use of hypercubes and naïve search was not enough to create scalable algorithms. On the other hand, the similarity search community has been tackling the neighborhood detection challenge in database systems with the similarity join operator. As it was shown in Section 2.2 from Chapter 2, there are three general approaches to perform a similarity join. All of them are potentially useful for outlier detection, as we demonstrated in the previous Chapter 4. It is interesting to note that the sorting procedure from the EGO family of join algorithms, i.e., the EGO-sort procedure, is a major contributor to perform join operations at scale.

With that in mind, this chapter introduces the second and last main contribution of this MSc work: one novel outlier detector named HySortOD that takes advantage of the EGO-sort idea to further speed up the detection without compromising accuracy. Additionally, we provide a default parameter configuration so to allow our algorithm to be truly unsupervised. Specifically, this chapter is organized as follows. In Section 5.1, we formally define the outlier detection problem investigated here. Section 5.2 introduces our novel algorithm HySortOD that takes advantage of hypercube sorting using a novel search strategy for fast and accurate detection. In Section 5.3, we describe our experimental setup and also report our findings. The discussions are in Section 5.4. Finally, the concluding remarks for the chapter are presented in Section 5.5.

5.1 Problem Statement

Likewise the problem statement of our ODSSJ algorithm, which was described in Section 4.2 from Chapter 4, here we also rely on the well-accepted assumption that instances with few neighbors are very likely outlying instances, and their closest instances may also be outliers. Once more, we consider that the dataset is represented in a d-dimensional space with m instances, and that a distance function is used to distinguish the instances. In contrast, instead of classifying instances as outliers and inliers, this chapter considers the use of an *outlierness score* that indicates the likelihood of each instance to be an outlier. Therefore, the neighborhood concept is formalized as in Definition 4, from the previous Chapter 4, while the additional concept of outlierness score is formally specified in the following with Definition 6.

Definition 6 (Outlierness Score)**.** The outlierness score of an instance q in a dataset X is given by the number of neighbors of q within radius ε normalized by the maximum number of neighbors existing for any other instance in dataset X. Formally, it is given by:

$$Outlierness(X,q,\varepsilon) = \frac{|rng(X,q,\varepsilon)|}{\max(\{|rng(X,x,\varepsilon)| \ : \ x \in X\})}$$

These definitions describe the exact solution for the outlier detection problem that we investigate in this chapter. The following section provides the details of our proposed algorithm.

5.2 The HySortOD Algorithm

This section presents our proposed algorithm HySortOD. In a nutshell, it 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, the EGO-sort procedure is used to organize the hypercubes in such a way that neighboring hypercubes with regard to the feature space get close to each other also in the array. Then, a novel 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 calculated, the last phase reports an outlierness score for each hypercube. Every instance that lies within a hypercube receives the same score – as it happens in virtually all hypercube-based algorithms found in the literature. The next sections detail our proposal.

5.2.1 Creating Hypercubes

The hypercubes are essentially bounded regions of the space where at least one instance exists. Without loss of generality, we assume that the dataset $X = [x_1, x_2, \ldots x_m]$ has m normalized instances, where each instance $x_p = [x_{p,1}, x_{p,2}, \ldots x_{p,d}]$ is within the d-dimensional hypercube $[0,1)^d$ and d is 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 hypercube side size is given by $l = \frac{1}{b}$, which means that the greater the value of b, the smaller are the hypercubes.

As it is shown in Algorithm 2, we represent the hypercubes in an 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 an instance $x_p \in X$ to hypercube coordinates as follows $h_i = [h_{i,1}, h_{i,2}, \ldots h_{i,d}] = [\lfloor x_{p,1}/l \rfloor, \lfloor x_{p,2}/l \rfloor, \ldots \lfloor x_{p,d}/l \rfloor]$. Each instance lies within only one hypercube boundaries - see Line 2 - and we denote as c_i the count of instances that lie within h_i - see Lines 5, 6 and 8. The counts are stored in an array $C = [c_1, c_2, \ldots c_n]$. The hypercubes are then used as box-countings to estimate their neighboring densities, as we describe latter in Section 5.2.3.

Algorithm 2 – `Create_hypercubes( )`

Input Dataset X; number of *bins* b.
Output Hypercubes H; Countings 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:          Create a new count c_i in C;
6:          c_i ← 0;
7:      end if
8:      c_i ← c_i + 1;
9:  end for
10: return H, C;
```

Intuitively, one can see this process as overlaying a *grid* into the data space to discretize and group instances. Figure 9a illustrates the mapping process of dataset instances to hypercube coordinates in a toy 2-dimensional dataset using $b = 4$, so the grid has up to 16 cells[1]. Note that empty hypercubes are *not* represented in memory *nor* processed, so the maximum number of hypercubes is always limited by the data cardinality.

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

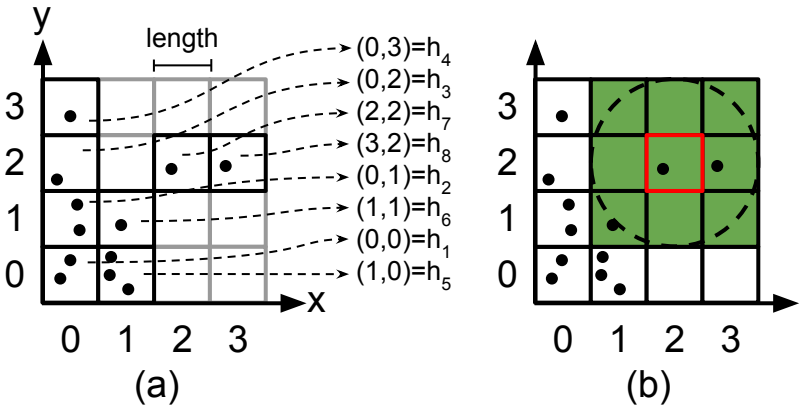


Figure 9 – (a) A *grid* using $b = 4$, and 8 *cells* with their coordinates. *Cells* in gray are not stored nor processed. (b) The immediate neighbors of *cell* $(2,2)$ are highlighted in green.

5.2.2 Sorting Hypercubes

Once the hypercube array H is created, the hypercubes are sorted considering their coordinates so that neighboring hypercubes with regard to the feature space get close to each other also in the array. In practice, it is one usual sorting procedure that considers d-dimensional hypercube coordinates in numerical order. For example, in Figure 9a, 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. The sorting procedure also does not require any advanced indexing structure still allowing an efficient approach in terms of space complexity. The following section presents how to take advantage of the sorted array by means of our novel neighborhood search strategy.

5.2.3 Neighborhood Search

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 search operations using hypercubes and their coordinates rather than dataset instances. Thus, for each hypercube $h_i \in H$, we spot its immediate neighbors based on Definition 7.

Definition 7 (Hypercube Neighborhood)**.** Given the hypercube array H and a hypercube h_i of interest, the hypercube neighborhood $N(h_i)$ includes all hypercubes with coordinates that are at no more than 1 unit of distance distant from the coordinates of h_i. That is: $N(h_i) = \{h_k \mid |h_{k,j} - h_{i,j}| \leq 1, \forall 1 \leq j \leq d; \forall h_k \in H\}$, where $h_{k,j}$ and $h_{i,j}$ are respectively the coordinates of hypercubes h_k and h_i in dimension j.

Note that Definition 7 can be seen as an approximation of a range search for dataset X with $\varepsilon = 3l/2$ and the query center being the centroid of h_i, as it is specified in Definition 4 from the previous Chapter 4, but the maximum distance between instances within the neighborhood

of h_i is actually $3l\sqrt{d}$. Definition 8 denotes the neighborhood density w_i of a hypercube h_i based on the number of instances that exist in its neighborhood $N(h_i)$. The densities are stored in an array $W = [w_1, w_2, \ldots, w_n]$.

Definition 8 (Neighborhood Density)**.** Given a hypercube h_i and its neighbors $N(h_i)$, the neighborhood density w_i is the count of instances that lie in h_i or in one of its neighbors. Formally, it is given by:

$$w_i = \sum_{h_k \in N(h_i)} c_k$$

Let us name as *prospective neighbor* of h_i any hypercube $h_k \in H$ that satisfies the condition $|h_{i,1} - h_{k,1}| \leq 1$. Similarly, we name as *immediate neighbor* of h_i any prospective neighbor of h_i that satisfies the full condition in Definition 7. Let us illustrate it considering the example sorted array $H = [h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8] = [(0,0), (0,1), (0,2), (0,3), (1,0), (1,1), (2,2), (3,2)]$ from Section 5.2.2 and a search operation centered at cell $h_7 = (2,2)$ from Figure 9. By sequentially scanning the sorted hypercubes in H, we can find the immediate neighbors $h_6 = (1,1)$, $h_7 = (2,2)$ and $h_8 = (3,2)$, thus, the neighborhood density is $w_7 = c_6 + c_7 + c_8 = 1 + 1 + 1 = 3$. Thanks to the sorting strategy, the prospective neighbors are already clustered together in array H. Note that the neighborhood search and the neighborhood density computation of a hypercube h_i are performed simultaneously.

However, during a linear scan many prospective neighbors would be tested which can increase the runtime. An ideal solution would minimize such tests and yet find all immediate neighbors. We observe that, it is expected to have a sequential range of hypercubes – 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 in 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 the algorithm to perform the density computation by simply traversing the appropriate branches of the tree instead of a full scan over array H. We describe the construction of the tree and the search algorithm in the following sections.

5.2.3.1 Construction

Algorithm 3 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 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. The root node encloses all hypercubes and does not map a coordinate value of any dimension,

that is: $P_{value} = \emptyset$, $P_{begin} = 1$ 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 respective dimension.

The algorithm starts by testing the two recursion base cases - see Lines 1 and 2, when the current dimension is greater 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 search and the granularity of the 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 position 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 position where the coordinate value is the same for all hypercubes within the sequence at dimension j - see Line 7. Next, we add the created node as a child of the current parent node P - see Line 8. Then, a recursion is called passing the child node to map the coordinate values of the next dimension - see Line 9. Finally, the beginning and 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 in Line 17.

Algorithm 3 – `Construct( )`

Input Hypercubes H; Value *MinSplit*; Parent node P; Column 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, $MinSplit$, 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, $MinSplit$, child node, $j+1$);
17: **return** P;

In Figure 10, we illustrate how our tree would be constructed based on the 2-dimensional dataset presented in Figure 9. On the left side, there is the root node that has no coordinate value associated with it mapping all existing sorted hypercubes in H, i.e., from position h_1 to h_8. 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., y), there are four child nodes for the coordinate value 0 of the first dimension because the *MinSplit* parameter is set to 4, which means that nodes mapping less than 4 hypercubes are not split into child nodes - see Line 2 in Algorithm 3. The nodes of the second dimension map only one hypercube each, as well as those nodes with coordinate values 1, 2 and 3 from the first dimension.

Note that the idea of constructing a dimension-wise tree is sensitive to the dimension ordering because some dimensions might provide more information gain than others, and potentially allow to prune more branches during the search step. 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 chapter, and we aim to investigate this point further in future work.

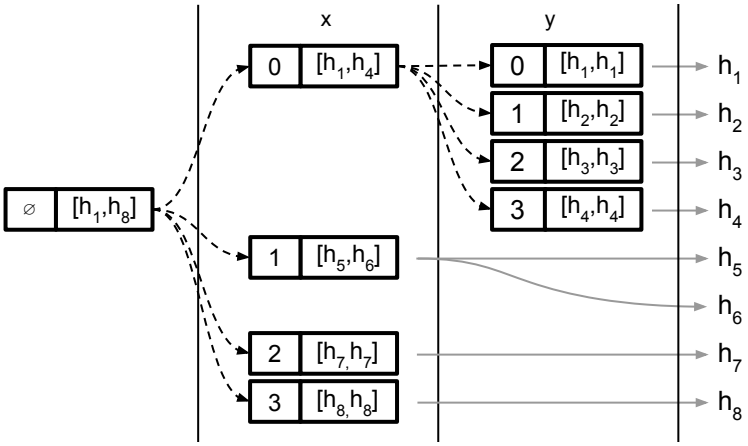


Figure 10 – Constructed tree mapping 8 hypercubes with $MinSplit = 4$. Dashed lines indicate the tree hierarchy and solid lines indicate the mapped hypercubes in array H.

5.2.3.2 Search

The search operation spots all immediate neighbors of a given hypercube and returns the total neighborhood density. Each level of the constructed tree represents a dimension and the child nodes of each level store the existing coordinate values with respect to the tree level, beginning and ending positions of prospective neighbors. Thus, the strategy is to traverse the tree branches and only scan over hypercubes mapped by the leaf nodes.

In Algorithm 4 the goal is to return the neighborhood density of a given hypercube. It starts by assigning the number of instances in 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 if the hypercube is an immediate neighbor; if so, we increment the total density - see Line 4. When P is not a leaf node, we recursively traverse

through the branches that are at most at 1 unit of distance distant from the given hypercube for each dimension - see Lines 7 to 9 and update the total density. Then, 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 4 – `Neighborhood_density( )`

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

```
1:  w_i ← c_i;
2:  if P is leaf then
3:      for k ← P_begin to P_end do
4:          if h_{i,j} is an immediate neighbour of h_{k,j} then w_i ← w_i + c_k;
5:      end for
6:  else
7:      for each node mapping the coordinates h_{i,j} ± 1 do
8:          w_i ← w_i + Neighborhood_density(H, C, node, i, j + 1);
9:      end for
10: end if
11: return w_i;
```

5.2.4 Outlierness Score

The final phase uses the neighborhood density of the hypercubes to assign an outlierness score to 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 6, but adjusted to the hypercube context, it becomes the ratio of the hypercube density to the maximum existing density, as it is shown in Definition 9.

Definition 9 (Hypercube-based Score)**.** Given a hypercube neighborhood density w_i and the maximum neighborhood density w_{max}, the outlierness score is defined as:

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

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 relative to the maximum neighborhood density $w_{max} = \max(W)$, where $W = [w_1, w_2, \ldots w_n]$ is the array with all neighborhood densities. The score values range in the closed-interval $[0, 1]$. High outlierness means scores close to 1, while low outlierness is represented by near-zero scores.

5.2.5 Proposed Algorithm

Algorithm 5 is the full pseudo-code of our proposed HySortOD. As we described before, it receives as input the dataset X, besides one intuitive parameter b that represents the number of *bins* to be created per dimension and the *MinSplit* value that sets the minimum number of hypercubes that 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. Latter in Section 5.4.4, we analyze the impact of b in the accuracy and the $MinSplit$ values in the runtime of HySortOD; also, we identify default values for both parameters that allow our proposal to be parameter-free and yet report fast high-quality results. Algorithm 5 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 Section 5.2.2. In Line 3, the hypercubes are mapped into a tree-based structure. Lines 4 to 9 compute the neighborhood densities W, and identify the largest one w_{max}. Finally, a data scan on X is performed in Lines 10 to 15 to obtain and report the outlierness scores O to the user - see Line 16.

5.2.5.1 Time Complexity

The total time complexity is the sum of the complexity of HySortOD 's individual phases. Time complexity of hypercube creation is $O(m)$ because we process only once each instance of dataset X to output the set of hypercubes of size n, where $n \leq m$. The sorting phase takes $O(dn \log n)$ time as 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 $O(b\frac{n}{b})$ time for the first dimension, and within each node in the first dimension, the child nodes of the second dimension takes $O(b\frac{n}{b^2})$ time, 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 its closed form $O(\frac{n(b-b^{1-d})}{b-1})$ or simply $O(3^d n)$. Since at least one instance lies within a hypercube region, the upper bound number of immediate neighbors is also limited by n, besides 3^d; that is, the maximum number of immediate neighbors is $s = min(\{3^d, n\})$. Thus, searching all immediate neighbors for all hypercubes takes $O(sn)$. To report the outlierness of each instance, when efficiently implemented, the process takes $O(m)$. Therefore, the final time complexity is $O(sn)$ when $sn > m$; otherwise, it is $O(m)$. Note that this analysis is based on the assumption that all instances are uniformly distributed in the data space, which is the worst-case scenario for *any* outlier detection algorithm, however, in practice such distribution is not expected to occur because the algorithm would not be able to distinguish outlying instances. After all, instances would be equidistant from each other. In our experiments, the worst-case scenario seems to have a much lower upper bound when analyzing the runtime results. The next section describes the experimental setup used to evaluate our algorithm.

Algorithm 5 – `HySortOD( )`

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

1: $H, C \leftarrow$ `Create_hypercubes(`X`,` b`);` ▷ Algorithm 2
2: Sort H and adjust C accordingly;
3: `Construct(`H, *MinSplit*, `root node, 1 );` ▷ Algorithm 3
4: Create empty density array W;
5: **for** each $h_i \in H$ **do**
6: $w_i \leftarrow$ `Neighborhood_density(`$H, C,$ `root node,` $i, 1$`);` ▷ Algorithm 4
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 9
14: Insert o_p into array O;
15: **end for**
16: **return** O;

5.3 Experimental Setup

This section presents our experimental setup. HySortOD was implemented in plain Java. We compare our proposal with 8 other algorithms: ODSSJ from the previous Chapter 4, which is also implemented in plain Java, and 7 state-of-the-art algorithms, i.e., kNN-Out, DB-Out, LOF, ODIN, HilOut, ABOD and aLOCI, which are coded in Java under the framework ELKI (ACHTERT *et al.*, 2011). For the related work's algorithms that can take advantage of an indexing data structure, we used ELKI's default in-memory hashtable that is expected to be faster than the other alternatives[2]. Also, we used the Euclidean distance for all applicable algorithms. To ensure a fair comparison among the competitors, we performed an exhaustive parameter search for each algorithm and dataset pair to find the highest result quality, which is reported. Appendices A, B and C present the best parameter configuration identified for each algorithm and dataset. **Observation:** for the purpose of reproducibility, all codes, detailed results, parameter values tested and datasets used in this chapter are freely available for download online[3,4].

Note that we report the results of two versions of our proposal. The first one, named HySortOD, uses a fixed parameter $b = 5$ as a default value for all datasets and we consider it to be our parameter-free proposal. *MinSplit* is set to 100 for all datasets because in our experiments it was able to reduce significantly the algorithm runtime. Latter in Section 5.4.4, we provide the details on how the default values were chosen. The second version, named HySortOD (Best),

[2] <https://elki-project.github.io/releases/current/doc/de/lmu/ifi/dbs/elki/database/StaticArrayDatabase.html>
[3] <https://github.com/eug/hysortod.java>
[4] <https://github.com/eug/hysortod.py>

uses the value of parameter b that maximizes the AUC score to allow a fair comparison with the other algorithms tested, since we also do this fine tuning for them.

In our experiments, we studied 12 labeled real-world benchmark datasets available in (RAYANA, 2016). These well-known datasets are commonly used by the outlier detection community to evaluate the quality and runtime results of the algorithms. We used the original version of these datasets and did not perform any form of pre-processing. Note that these are the same datasets that we studied in the previous Chapter 4, which are summarized in Table 2. We decided to split the datasets into two groups: (a) small datasets – the first 10 datasets in Table 2, except for *satimage2*, and; (b) large datasets – *satimage2*, *shuttle* and *http*. The groups were defined with respect to the data cardinality and dimensionality. Note that *satimage2* is considered to be large due to its largest dimensionality and medium-sized cardinality.

For the effectiveness evaluation, we used the well-accepted ROC AUC (AUC) metric, which ranges between 0 and 1, where a perfect result is scored 1 and a random result is around 0.5. For the efficiency evaluation, we conducted 10 independent executions of each algorithm and dataset pair, recording the total runtime in seconds, so we always report average and standard deviation results. Note that we do not record the time spent to find the best parameter for each algorithm, which is a time-consuming task based on our experience. All experiments were performed in GNU/Linux Ubuntu Xenial x86-64 running on a machine with processor Intel© Core™ i7-2600S @ 2.80GHz x 4 cores and 8GB of RAM.

5.4 Results and Discussion

This section reports and discusses the results of our experimental evaluation under the setup of Section 5.3. We aimed at answering the questions as follows:

Q1 Compared with 8 other algorithms, how efficient and effective is our proposed HySortOD on benchmark datasets?

Q2 How scalable are these techniques?

Q3 What are the effects of varying our parameters b and $MinSplit$, and how to obtain highly-accurate results from HySortOD in a parameter-free fashion?

Q4 Compared with 8 other algorithms, how well our HySortOD performs when used as an out-of-the-box solution for a real-world, non-benchmark problem?

5.4.1 Effectiveness Evaluation

This section investigates Question **Q1**: *"Compared with* 8 *other algorithms, how efficient and effective is our proposed HySortOD on benchmark datasets?"*. Table 6 summarizes the quality-related results for all algorithms and datasets. As it can be seen, ABOD reported the highest scores in 3 datasets, but it failed to run in the two largest datasets *shuttle* and *http* because of exceeded main memory capacity. Similarly, HilOut and aLOCI also exceeded the main memory available for dataset *http*. In our opinion, none of the datasets studied is large enough to justify a need for more than 8 GB of RAM. *k*NN-Out reported the highest score in 2 datasets, while LOF and HySortOD (Best) reported the highest score in 1 dataset each. Finally, DB-Out, ODIN and aLOCI did not report the highest score in any dataset.

Aside from the related works' algorithms, our approach ODSSJ reported similar results as HySortOD , being able to obtain the highest score among all other algorithms in 4 datasets. These high-quality results of ODSSJ can be justified by its *exact* approach on counting the number of neighbors combined with a fine-tuned threshold. The minor differences from ODSSJ 's best quality results compared with those of HySortOD indicate that our approximation strategy is well-suited to detect outliers effectively. Note that our HySortOD (Best) reported the highest average result overall compared with the other 8 algorithms. Additionally, its parameter-free version HySortOD reported similar results compared with the best algorithms using its default parametrization, which shows a significant advantage in real-world applications by not requiring any parameter tuning.

5.4.2 Efficiency Evaluation

This section further investigates Question **Q1**: *"Compared with* 8 *other algorithms, how efficient and effective is our proposed HySortOD on benchmark datasets?"*. Now, we are focused on efficiency. Table 7 summarizes the runtime results obtained from all algorithms and datasets. As it can be seen, both versions of our algorithm outperformed all related works' algorithms in terms of runtime, obtaining expressive efficiency gains in nearly all datasets. The runtime difference between our versions HySortOD (Best) and HySortOD was insignificant in all cases. Note that the overall average results of HilOut, aLOCI and ABOD do not consider all datasets, since they failed in the largest ones, but still our proposal presented considerable efficiency improvements regarding the average runtime. Algorithms *k*NN-Out, DB-Out, LOF and ODIN reported high runtime values for datasets larger than $40k$ instances. Also, ABOD had huge runtime requirements for any dataset larger than $1k$ instances. It makes these algorithms impractical for many real-world applications. Nevertheless, when comparing HySortOD with our algorithm ODSSJ , we note that both report similar results for small datasets while in the larger ones the runtime of ODSSJ starts to increase significantly, and yet, it is still very efficient compared with the state-of-the-art algorithms.

Figure 11 shows the big picture regarding both efficiency and effectiveness of all state-of-the-art algorithms considered in our study, thus truly answering Question **Q1**, where a fictitious, ideal algorithm would obtain perfect quality instantly. As it can be seen, our proposed HySortOD reported similar quality (AUC) scores in small and large datasets compared with 7 state-of-the-art algorithms, still being the fastest algorithm in *every single case* with improvements that refer to 3 or 4 *orders of magnitude* in many cases. Also, note that these results were obtained using our fixed default value for parameters b and $MinSplit$, i.e., the results of HySortOD (Best) are *not* shown in Figure 11. This is a massive advantage over the other algorithms studied when processing large real-world datasets that do not have labels available.

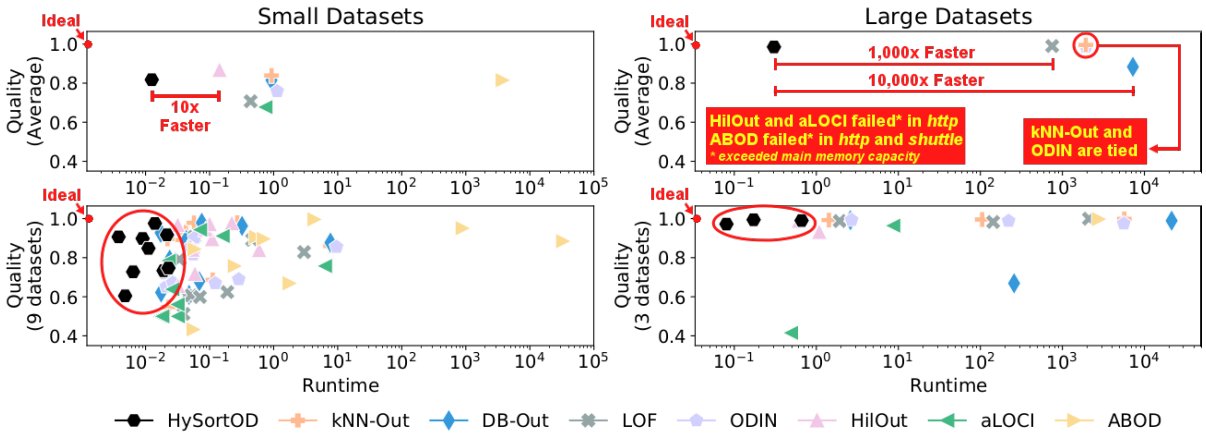


Figure 11 – Quality of results vs. runtime in *log* scale for 12 real-world datasets and 7 algorithms. Results from small and large datasets are presented in separate, respectively on the left and the right sides. Top: average of results. Bottom: detailed results. Our HySortOD consistently outperformed 7 state-of-the-art algorithms, being up to 4 *orders of magnitude* faster, while 3 competitors failed in large datasets.

5.4.3 Scalability Evaluation

This section investigates Question **Q2**: *"How scalable are the techniques studied?"*. One of the main challenges in outlier detection is scalability to process large datasets. To the best of our knowledge, there is no labeled dataset for outlier detection that is larger in cardinality than *http*. For this reason, we decided to demonstrate the scalability of our HySortOD by further investigating this dataset. Specifically, we created and processed smaller versions of *http* with random samples of increasing sizes up to the full dataset, *i.e.*, 1%, 10%, 15%, 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. Figure 12 reports the corresponding results for HySortOD and for the 7 state-of-the-art competitors studied. Regarding the related work, the results show a superlinear trend in virtually all cases. ABOD ran successfully with 1% sample size, but it took a massive runtime that exceeds the plot area; the main memory was not enough for samples larger than 10%. aLOCI and HilOut exceeded the main memory capacity after 15% and 75% sample sizes, respectively. The remaining algorithms required massive amounts of time to complete.

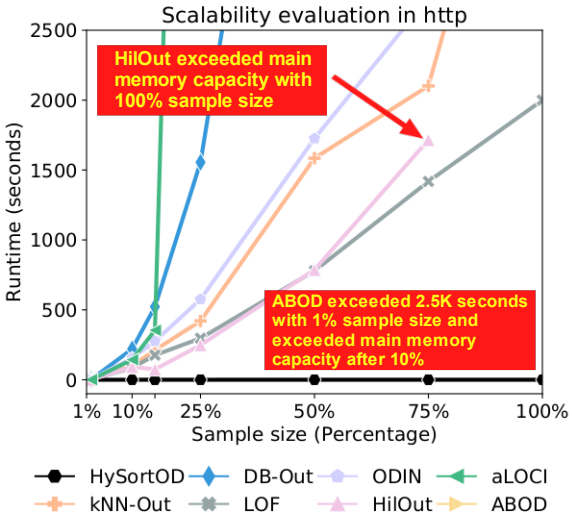


Figure 12 – Runtime on random samples from *http*. Our HySortOD scales much better than 7 competitors.

In contrast, our HySortOD was able to keep a much better scalability on the sample size. We conducted an in-depth analysis of these results and noted that distinguishing outlier and inlier instances in *http* is relatively easy, since they are commonly far away from each other. Therefore, our HySortOD only created around 20 hypercubes for this dataset, which is a huge advantage compared with other algorithms that needed to perform neighborhood searches in more than $500k$ instances. These findings suggest that the use of our proposed hypercube-ordering-and-searching strategy to detect outliers is indeed a powerful scale-up tool.

5.4.4 Parametrization

This section investigates Question **Q3**: *"What are the effects of varying our parameters b and MinSplit, and how to obtain highly-accurate results from HySortOD in a parameter-free fashion?"*. The general intuition behind our parameter b is that, as the number of bins increases, the hypercube length $l = \frac{1}{b}$ decreases, and so does the neighborhood radius. Therefore, b indeed influences the quality of HySortOD. For large values of b, it is expected all instances to be reported as outliers, because they will lie alone in their hypercube neighborhood. By contrast, it is expected for small values of b that all instances may be reported as inliers, because the hypercubes will likely be close enough to make it impossible to distinguish between inliers and outliers. To study the effect of b in the effectiveness, we varied its value from 2 up to 40 for all 12 datasets. Figure 13 reports our findings; it plots the AUC score versus b. Note that the horizontal axis is in log scale to improve the visualization. We observe three distinct intervals with different patterns in the horizontal axis. We name the intervals according to their general quality trend, that is: unstable, steady, and decreasing quality in the intervals $[2,3]$, $[4,7]$, and $[8,40]$, 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 12 real datasets of distinct domains, so we suggest $b = 5$ to be the default configuration of HySortOD.

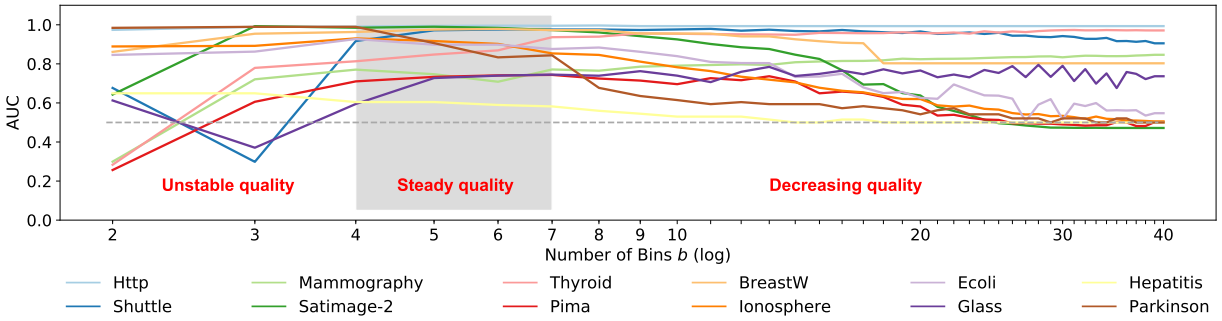


Figure 13 – AUC score (quality) vs. number of bins b for 12 real datasets. There are three distinct intervals with different patterns; the best results are likely to be in the steady quality interval, so we suggest the use of $b = 5$.

Additionally, we analyzed different variations for our *MinSplit* parameter, which only affect runtime; not accuracy. It balances the number of hypercubes mapped by a node, and, thus, limits the number of hypercubes to scan during the neighborhood search. In general, small values tend to leave the leaf nodes mapping only a few hypercubes, while large values have the opposite effect, which tends to degenerate to the linear scan runtime. With that in mind, we conducted experiments varying the parameter from 10 to 1,000 in each of 12 benchmark datasets, and according to our analysis we observe that setting the value 100 generally reduces the runtime when compared to other values.

5.4.5 Case Study: Breast Cancer Detection

This section investigates Question **Q4**: *"Compared with 7 state-of-the-art algorithms, how well our HySortOD performs when used as an out-of-the-box solution for a real-world, non-benchmark problem?"*. Aside from benchmark datasets, we also experimented HySortOD in the important real-world problem of breast cancer detection, where the goal is to detect the malignant cases. This problem has been traditionally framed as a supervised imbalanced classification problem, however, as the malignant cases are often rare, it makes a suitable scenario to be framed as an outlier detection problem. Thus, for this experiment, we considered the KDD Cup 2008[5] dataset that consists of 102,294 regions of interest screen images; each image is represented by 117 features, without description, and a label indicating if the image has the presence of cancer (malignant) or not (benign).

We conducted the experiment as described in Section 5.3. Our algorithms HySortOD and HySortOD (Best) were evaluated in terms of quality and runtime. They reported the AUC score of 0.6097 (using the default parameter $b = 5$) and 0.6389 (using parameter $b = 7$), respectively. The average runtime and standard deviation for our algorithms were $5\text{x}10^2 \pm 95.63$ and $3\text{x}10^2 \pm 20.23$ seconds, respectively. The related works' algorithms were able to process the dataset, except

[5] <https://www.kdd.org/kdd-cup/view/kdd-cup-2008/Data>

aLOCI and ABOD (exceeded main memory capacity), and the average and standard deviation runtime over all parameter attempts was $2\text{x}10^3 \pm 7\text{x}10^4$, however, they were unable to detect outliers. In the parameter search for this dataset, we varied the parameter k (kNN-Out, LOF, ODIN and HilOut) and d (DB-Out) from 0 to $10,000$ with step of 50. For each algorithm, we conducted 200 parameters attempts totaling roughly $4\text{x}10^5$ seconds of runtime.

These results are a strong evidence that our parameter-free algorithm HySortOD shows a superior advantage compared with existing algorithms in two aspects. First, our algorithm was able to detect outliers while the existing algorithms were unable to do so. Second, there is no need to fully depend on specialist expertise and spend time in parameter-tuning to report satisfactory results. Thus, in practice, we observe that our proposed algorithm can be a powerful outlier detector not only by its efficiency due to its hypercube ordering but also for its practicality due to the parameter-free approach.

5.5 Conclusion

This chapter presented the new algorithm HySortOD for outlier detection. Our main contributions are:

C1 Speed and Scalability: We carefully designed HySortOD to efficiently distinguish inliers and outliers by means of a new hypercube-ordering-and-searching approach that speeds-up the outlier detection task to perform at scale;

C2 Simplicity: We provide default parameter values that allow HySortOD to be truly unsupervised, *i.e.*, to be parameter-free, and yet deliver insightful results, which can save significant amount of manual work in the parameter-tuning task;

C3 Benchmark Experiments: We study 12 benchmark datasets from distinct domains with up $\sim$500k instances, and show that our proposal produces highly accurate results with one speed-up of up to 4 *orders of magnitude* over 7 state-of-the-art algorithms, even when it is used in a parameter-free fashion;

C4 Case Study Experiments: We study the crucial task of breast cancer detection to show that our approach can be successfully used as an out-of-the-box solution for real-world, non-benchmark problems. This study used a well-known dataset with $\sim$100k instances and $\sim$120 dimensions that was originally created for supervised, imbalanced classification.

Finally, note that our solution does not require parameter-tuning to detect outliers, but some applications might require additional steps to obtain relevant insights from the results, like to specify a subset of top outliers of interest or a threshold value for outlierness scores.

Dataset	HySortOD	HySortOD (Best)	ODSSJ	kNN-Out	DB-Out	LOF	ODIN	*HilOut	*aLOCI	*ABOD
parkinson	0.9167	**0.9896**	**0.9896**	0.8958	0.9271	0.7917	0.7188	0.9688	0.5000	0.8438
hepatitis	0.6045	0.6495	0.6493	0.5473	0.6219	0.5124	0.6517	**0.6567**	0.5000	0.4328
glass	0.7274	0.7951	**0.8488**	0.7824	0.7897	0.6043	0.6724	0.8423	0.6382	0.7572
ecoli	0.8981	0.9271	**0.9557**	0.8899	0.6096	0.6070	0.9018	0.8964	0.7851	0.9059
ionosphere	0.9168	0.9304	0.9108	0.9314	0.9066	0.8606	0.8526	**0.9467**	0.5000	0.8961
breastw	0.9754	0.9789	0.9632	0.9782	0.9785	0.5989	0.8166	0.9730	0.9430	**0.9960**
pima	0.7334	0.7435	**0.7973**	0.6864	0.6790	0.6240	0.6692	0.7185	0.5608	0.6688
thyroid	0.8477	0.9748	0.8182	**0.9793**	0.9624	0.8913	0.6896	0.9779	0.9110	0.9508
satimage2	0.9901	0.9931	0.9904	0.9924	0.9919	0.9875	0.9931	0.9852	0.4152	**0.9972**
mammography	0.7470	0.8464	0.8071	0.8597	0.8772	0.8285	0.8551	0.8405	0.7572	**0.8844**
shuttle	0.9718	0.9792	0.9054	**0.9953**	0.6688	0.9827	0.9883	0.9335	0.9638	-
http	0.9941	0.9966	0.9892	0.9987	0.9891	**0.9995**	0.9746	-	-	-
Average	0.8603 ± 0.13	**0.9003 ± 0.12**	0.8969 ± 0.09	0.8777 ± 0.14	0.8335 ± 0.15	0.7740 ± 0.18	0.8318 ± 0.13	$*0.8950 \pm 0.10$	$*0.6795 \pm 0.20$	$*0.8333 \pm 0.17$

*The average values only consider reported measurements

Table 6 – AUC scores (quality) for 8 algorithms and 12 benchmark datasets. Best results are highlighted in bold. Note that our proposed HySortOD is *very accurate*.

Dataset	HySortOD	HySortOD (Best)	ODSSJ	kNN-Out	DB-Out	LOF	ODIN	*HilOut	*aLOCI	*ABOD
parkinson	**0.01 ± 0.01**	**0.01 ± 0.01**	**0.01 ± 0.01**	0.02 ± 0.01	0.02 ± 0.01	0.03 ± 0.01	0.02 ± 0.01	0.03 ± 0.01	0.02 ± 0.01	0.06 ± 0.01
hepatitis	**0.01 ± 0.01**	**0.01 ± 0.01**	**0.01 ± 0.01**	0.02 ± 0.01	0.02 ± 0.01	0.04 ± 0.01	0.02 ± 0.01	0.03 ± 0.01	0.02 ± 0.01	0.06 ± 0.01
glass	**0.01 ± 0.01**	**0.01 ± 0.01**	**0.01 ± 0.01**	0.03 ± 0.01	0.02 ± 0.01	0.04 ± 0.01	0.03 ± 0.01	0.05 ± 0.01	0.03 ± 0.01	0.25 ± 0.03
ecoli	**0.01 ± 0.01**	**0.01 ± 0.01**	**0.01 ± 0.01**	0.04 ± 0.01	0.05 ± 0.01	0.05 ± 0.01	0.06 ± 0.01	0.10 ± 0.01	0.02 ± 0.01	0.54 ± 0.02
ionosphere	**0.03 ± 0.01**	**0.03 ± 0.01**	**0.02 ± 0.01**	0.05 ± 0.01	0.04 ± 0.01	0.06 ± 0.02	0.05 ± 0.01	0.09 ± 0.01	0.03 ± 0.01	0.71 ± 0.01
breastw	**0.01 ± 0.01**	**0.01 ± 0.01**	**0.01 ± 0.01**	0.06 ± 0.01	0.07 ± 0.01	0.08 ± 0.02	0.06 ± 0.01	0.10 ± 0.01	0.07 ± 0.01	1.80 ± 0.13
pima	**0.02 ± 0.01**	**0.02 ± 0.01**	**0.02 ± 0.01**	0.11 ± 0.02	0.07 ± 0.01	0.19 ± 0.03	0.12 ± 0.02	0.06 ± 0.01	0.03 ± 0.14	4.40 ± 0.19
thyroid	**0.04 ± 0.01**	0.04 ± 0.01	0.04 ± 0.01	0.27 ± 0.01	0.32 ± 0.02	0.46 ± 0.03	0.28 ± 0.01	0.22 ± 0.01	0.16 ± 0.01	$1x10^3 \pm 29.29$
satimage2	1.04 ± 0.01	1.39 ± 0.03	**0.14 ± 0.01**	1.42 ± 0.05	2.61 ± 0.07	1.93 ± 0.10	2.69 ± 0.04	0.61 ± 0.01	0.49 ± 0.02	$3x10^3 \pm 1x10^2$
mammography	**0.06 ± 0.01**	0.18 ± 0.01	0.17 ± 0.02	7.76 ± 1.09	7.70 ± 0.72	2.98 ± 0.26	9.59 ± 0.76	0.59 ± 0.07	6.26 ± 0.10	$3x10^4 \pm 1x10^4$
shuttle	0.18 ± 0.02	**0.17 ± 0.02**	3.11 ± 0.07	$1x10^2 \pm 0.78$	$2x10^2 \pm 3.87$	$1x10^2 \pm 9.05$	$2x10^2 \pm 2.31$	1.08 ± 0.05	8.59 ± 0.13	-
http	0.23 ± 0.03	**0.28 ± 0.02**	27.14 ± 0.85	$5x10^3 \pm 6x10^2$	$2x10^4 \pm 7x10^2$	$2x10^3 \pm 63.90$	$5x10^3 \pm 1x10^2$	-	-	-
Average	**0.14±0.29**	0.19 ± 0.39	2.55 ± 7.45	$4x10^2 \pm 1x10^3$	$2x10^3 \pm 6x10^3$	$1x10^2 \pm 5x10^2$	$3x10^2 \pm 1x10^3$	*0.27 ± 0.34	*1.43 ± 3.01	$*4x10^3 \pm 1x10^4$

*The average values only consider reported measurements

Table 7 – Runtime in seconds for 8 algorithms and 12 benchmark datasets. Best results are highlighted in bold. Note that our HySortOD is commonly 3 to 4 *orders of magnitude* faster than the others.

CHAPTER

6

CONCLUSION

In this work, we aim at improving the efficiency and scalability of outlier detection due its struggle to process large volumes of data. To develop our research towards this goal, we make two fundamental observations in similarity join operation that are the source of inspiration for this work. First, we observe that similarity join is designed to operate in database systems, where efficiency in data processing is crucial. Second, we discuss and demonstrate that outlier detection and similarity join operation share theoretical concepts that allow the former to employ existing techniques from the latter. To the best of our knowledge no other work has explicitly investigated the link between these tasks. Based on these observations, we proposed two novel outlier detection algorithms - ODSSJ (Chapter 4) and HySortOD (Chapter 5) - that both report similar quality results and are generally 3 *orders of magnitude faster* than existing algorithms in the literature. Therefore, we highlight four major contributions of our research:

C1 **Generality**: As seen in our experiments (Sections 4.4, 5.3 and 5.4.5 from Chapters 4 and 5) we conducted a comprehensive evaluation of our proposed algorithms in a wide range of applications. It shows that the algorithm's assumptions are not domain specific, therefore being able to support many types of business decisions;

C2 **Simplicity**: The proposed algorithms offer simplifications in terms of implementation and usage. The adaptation of ODSSJ for other settings beyond in-memory single-thread could be simplified due the use of the well-known similarity join operation. The lack of specialist-defined parameters for HySortOD simplifies its usage by saving time during the parameter-tuning part;

C3 **Interpretability**: Both algorithms are designed under the well-accepted and interpretable assumption of neighborhood. It facilitates the results communication especially for inter-disciplinary teams that do not contain the expertise in data mining;

C4 **Speed**: Our experimental results (Sections 4.5 and 5.4 from Chapters 4 and 5) report a substantial gain in terms of runtime compared to other existing algorithms (Tables 4 and 7 from Chapters 4 and 5). These results are consistent for both proposals, therefore supporting the hypothesis of this work and significantly contributing for fast outlier detection.

Finally, let us highlight that this MSc work generated the following publication:

- Eugênio F. Cabral, Robson L. F. Cordeiro: Fast and Scalable Outlier Detection with Sorted Hypercubes. In: 29th ACM International Conference on Information and Knowledge Management — CIKM, 2020, Virtual Event, Ireland. p. 95-104, DOI:10.1145/3340531.3412033, **Full paper at an International Conference Qualis-CC A1**.

BIBLIOGRAPHY

ACHTERT, E.; HETTAB, A.; KRIEGEL, H.-P.; SCHUBERT, E.; ZIMEK, A. Spatial Outlier Detection: Data, Algorithms, Visualizations. In: **International Symposium on Spatial and Temporal Databases**. Minneapolis, MN, USA: Springer, 2011. p. 512–516. Citations on pages 48, 58, and 76.

ADHAV, L. R.; KUMAR, S. D. MVJoin: An Efficient Approach for Record Linkage and Duplication Finding. In: **International Conference on Emerging Trends in Computer Engineering**. Kobe, Japan: [s.n.], 2015. p. 2277–9477. Citations on pages 38 and 49.

AGGARWAL, C. C. **Outlier Analysis**. New York: Springer, 2017. 445 p. Citations on pages 29, 32, 53, 54, and 57.

ANBARASI, M. S.; DHIVYA, S. Fraud Detection Using Outlier Predictor in Health Insurance data. In: **International Conference on Information, Communication & Embedded Systems**. Chennai, India: IEEE, 2017. p. 6. Citation on page 29.

ANGIULLI, F.; PIZZUTI, C. Fast Outlier Detection in High Dimensional Spaces. In: **European Conference on Principles of Data Mining and Knowledge Discovery**. Helsinki, Finland: Springer, 2002. p. 15–27. Citation on page 47.

AYADI, A.; GHORBEL, O.; OBEID, A. M.; ABID, M. Outlier Detection Approaches for Wireless Sensor Networks: A Survey. **Computer Networks**, Elsevier, v. 129, p. 319–333, 2017. Citation on page 29.

BARNETT, V.; LEWIS, T. **Outliers in Statistical Data**. New York: Wiley, 1994. Citation on page 29.

BELLMAN, R. E. **Adaptive Control Processes: A Guided Tour**. Princeton, New Jersey, USA: Princeton University Press, 1961. 255 p. Citations on pages 36 and 47.

BERCHTOLD, S.; KEIM, D. A.; KRIEGEL, H.-P. The X-tree: An Index Structure for High-Dimensional Data. In: **VLDB Conference**. Mumbai, India: ACM, 1996. p. 28–39. Citation on page 46.

BINDU, P. V.; THILAGAM, P. S.; AHUJA, D. Discovering Suspicious Behavior in Multilayer Social Networks. **Computers in Human Behavior**, Elsevier, v. 73, p. 568–582, 2017. Citation on page 29.

BÖHM, C.; BRAUNMÜLLER, B.; BREUNIG, M.; KRIEGEL, H.-P. High Performance Clustering Based on the Similarity Join. In: **International Conference on Information and Knowledge Management**. McLean, Virginia, USA: ACM, 2000. p. 298–305. Citation on page 49.

BÖHM, C.; BRAUNMÜLLER, B.; KREBS, F.; KRIEGEL, H.-P. Epsilon Grid Order : An Algorithm for the Similarity Join on Massive High-Dimensional Data. In: **International Conference on Management of Data**. Santa Barbara, CA, USA: ACM SIGMOD, 2001. v. 30, n. 2, p. 379–388. Citations on pages 42 and 49.

BREUNIG, M. M.; KRIEGEL, H.-P.; NG, R. T.; SANDER, J. LOF: Identifying Density-Based Local Outliers. In: **International Conference on Management of Data**. Dallas, TX, USA: ACM SIGMOD, 2000. v. 29, n. 2, p. 93–104. Citations on pages 26, 37, and 46.

BRINKHOFF, T.; KRIEGEL, H.-P.; SEEGER, B. Efficient Processing of Spatial Joins Using R-trees. **International Conference on Management of Data**, ACM, v. 22, n. 2, p. 237–246, 1993. Citation on page 41.

BRYAN, B.; EBERHARDT, F.; FALOUTSOS, C. Compact Similarity Joins. In: **International Conference on Data Engineering**. Cancun, Mexico: IEEE, 2008. p. 346–355. Citation on page 49.

CAMPELLO, R. J. G. B.; MOULAVI, D.; ZIMEK, A.; SANDER, J. Hierarchical Density Estimates for Data Clustering, Visualization, and Outlier Detection. **Transactions on Knowledge Discovery from Data**, ACM, v. 10, n. 1, p. 51, 2015. Citation on page 34.

CAMPOS, G. O.; ZIMEK, A.; SANDER, J.; CAMPELLO, R. J.; MICENKOVÁ, B.; SCHUBERT, E.; ASSENT, I.; HOULE, M. E. On the evaluation of unsupervised outlier detection: measures, datasets, and an empirical study. **Data Mining and Knowledge Discovery**, Springer, v. 30, n. 4, p. 891–927, 2016. Citations on pages 26, 55, 58, and 59.

CHANDOLA, V.; BANERJEE, A.; KUMAR, V. Anomaly Detection: A Survey. **Computing Surveys**, ACM, v. 41, n. 3, p. 1–58, 2009. Citations on pages 34, 37, and 53.

CHE, D.; SAFRAN, M.; PENG, Z. From Big Data to Big Data Mining: Challenges, Issues, and Opportunities. In: **International Conference on Database Systems for Advanced Applications**. Wuhan, China: Springer, 2013. p. 1–15. Citation on page 25.

CHEN, Z.; KALASHNIKOV, D. V.; MEHROTRA, S. Exploiting Context Analysis for Combining Multiple Entity Resolution Systems. In: **International Conference on Management of Data**. Providence, Rhode Island, USA: ACM SIGMOD, 2009. p. 207–218. Citations on pages 38 and 49.

CULOTTA, A.; MCCALLUM, A. Joint deduplication of multiple record types in relational data. In: **International Conference on Information and Knowledge Management, Proceedings**. Bremen, Germany: ACM, 2005. p. 257–258. Citation on page 49.

DHEERU, D.; TANISKIDOU, E. K. **UCI Machine Learning Repository**. 2017. Available: <http://archive.ics.uci.edu/ml>. Citation on page 58.

DITTRICH, J.-P.; SEEGER, B. GESS: A Scalable Similarity-Join Algorithm for Mining Large Data Sets in High Dimensional Spaces. In: **International Conference on Knowledge Discovery and Data Mining**. San Francisco, California, USA: ACM SIGKDD, 2001. p. 47–56. Citation on page 42.

DOHNAL, V.; GENNARO, C.; ZEZULA, P. Similarity Join in Metric Spaces Using eD-Index. In: **International Conference on Database and Expert Systems Applications**. Prague, Czech Republic: Springer, 2003. p. 484–493. Citation on page 54.

DONG, X.; HALEVY, A.; MADHAVAN, J. Reference reconciliation in complex information spaces. In: **International Conference on Management of Data**. Baltimore, Maryland, USA: ACM SIGMOD, 2005. p. 85–96. Citation on page 49.

DU, Q.; LI, X. A novel KNN join algorithms based on Hilbert R-tree in MapReduce. In: **International Conference on Computer Science and Network Technology**. Dalian, China: IEEE, 2013. p. 417–420. Citations on pages 40 and 41.

EMMOTT, A.; DAS, S.; DIETTERICH, T.; FERN, A.; WONG, W.-K. A Meta-Analysis of the Anomaly Detection Problem. **arXiv preprint**, arXiv, p. 35, 2015. Available: <https://arxiv.org/abs/1503.01158>. Citation on page 58.

ESTER, M.; KRIEGEL, H.-P.; SANDER, J.; XU, X. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In: **International Conference on Knowledge Discovery and Data Mining**. Portland, Oregon, USA: ACM, 1996. p. 226–231. Citations on pages 34 and 49.

FINKEL, R. A.; BENTLEY, J. L. Quad Trees A Data Structure for Retrieval on Composite Keys. **Acta Informatica**, Springer, v. 4, n. 1, p. 1–9, 1974. Citation on page 45.

FREDRIKSSON, K.; BRAITHWAITE, B. Quicker range- and k-NN joins in metric spaces. **Information Systems**, Elsevier, v. 52, p. 189–204, 2015. Citation on page 54.

FU, Y.; AGGARWAL, C.; PARTHASARATHY, S.; TURAGA, D. S.; XIONG, H. REMIX: Automated Exploration for Interactive Outlier Detection. In: **International Conference on Knowledge Discovery and Data Mining**. Halifax, Nova Scotia, Canada: ACM SIGKDD, 2017. p. 827–835. Citation on page 48.

GAO, J.; TAN, P.-N. Converting Output Scores From Outlier Detection Algorithms Into Probability Estimates. In: **International Conference on Data Mining**. Hong Kong: IEEE, 2006. p. 1–10. Citation on page 33.

GHOTING, A.; PARTHASARATHY, S.; OTEY, M. E. Fast Mining of Distance-based Outliers in High-dimensional Datasets. **Data Mining and Knowledge Discovery**, Springer, v. 16, n. 3, p. 349–364, 2008. Citation on page 38.

GIANNAKOPOULOU, S.; KARPATHIOTAKIS, M.; GAIDIOZ, B.; AILAMAKI, A. CleanM: An Optimizable Query Language for Unified Scale-out Data Cleaning. In: **VLDB Conference**. Munich, Germany: ACM, 2017. v. 10, n. 11, p. 1466–1477. Citations on pages 38 and 49.

GOLDSTEIN, M.; UCHIDA, S. A Comparative Evaluation of Unsupervised Anomaly Detection Algorithms for Multivariate Data. **PLoS ONE**, PLOS, v. 11, n. 4, p. 1–31, 2016. Citations on pages 26 and 55.

HAUTAMÄKI, V.; KÄRKKÄINEN, I.; FRÄNTI, P. Outlier Detection Using k-Nearest Neighbour Graph. In: **International Conference on Pattern Recognition**. Cambridge, UK: IEEE, 2004. p. 430–433. Citation on page 46.

HAWKINS, D. M. **Identification of Outliers**. 1. ed. New York, USA: Springer, 1980. ISBN 978-94-015-3996-8,978-94-015-3994-4. Citations on pages 26 and 29.

HE, Z.; XU, X.; DENG, S. Discovering Cluster-based Local Outliers. **Pattern Recognition Letters**, v. 24, n. 9-10, p. 1641–1650, 2003. Citation on page 34.

HOULE, M. E.; KRIEGEL, H.-P.; KRÖGER, P.; SCHUBERT, E.; ZIMEK, A. Can Shared-Neighbor Distances Defeat the Curse of Dimensionality? In: **International Conference on Scientific and Statistical Database Management**. Heidelberg, Germany: Springer, 2010. p. 482–500. Citation on page 48.

ISHIKAWA, Y.; TSUDA, K.; SADAKANE, K.; WANG, W.; QIN, J.; XIAO, C. Efficient Error-Tolerant Query Autocompletion. In: **VLDB Conference**. Riva del Garda, Trento, Italy: ACM, 2013. v. 6, n. 6, p. 373–384. Citation on page 38.

JABEZ, J.; MUTHUKUMAR, B. Intrusion Detection System (ids): Anomaly Detection Using Outlier Detection Approach. In: **International Conference on Intelligent Computing, Communication & Convergence**. Bhubaneswar, Odisha, India: Elsevier, 2015. p. 338–346. Citation on page 29.

JAUHRI, A.; MCDANEL, B.; CONNOR, C. Outlier Detection for Large Scale Manufacturing Processes. In: **Big Data 2015**. Santa Clara, CA, USA: IEEE, 2015. p. 2771–2774. Citation on page 29.

JIA, L.; ZHANG, L.; YU, G.; YOU, J.; DING, J.; LI, M. A Survey on Set Similarity Search and Join. **International Journal of Performability Engineering**, RAMS Consultants, v. 14, n. 2, p. 245–258, 2018. Citation on page 40.

JIN, W.; TUNG, A. K. H.; HAN, J.; WANG, W. Ranking Outliers Using Symmetric Neighborhood Relationship. In: **Pacific-Asia Conference on Knowledge Discovery and Data Mining**. Singapore: Springer, 2006. p. 577–593. Citation on page 47.

KALASHNIKOV, D. V. Super-EGO: Fast Multi-dimensional Similarity Join. **VLDB Journal**, ACM, v. 22, n. 4, p. 561–585, 2013. Citations on pages 43 and 54.

KALASHNIKOV, D. V.; MEHROTRA, S. Domain-independent data cleaning via analysis of entity-relationship graph. **Transactions on Database Systems**, ACM, v. 31, n. 2, p. 716–767, 2006. Citation on page 49.

KALASHNIKOV, D. V.; PRABHAKAR, S. Fast Similarity Join for Multi-Dimensional Data. **Information Systems**, Elsevier, v. 32, n. 1, p. 160–177, 2007. Citations on pages 43 and 49.

KELLER, F.; MÜLLER, E.; BÖHM, K. HiCS: High Contrast Subspaces for Density-based Outlier Ranking. In: **International Conference on Data Engineering**. Washington, DC, USA: IEEE, 2012. p. 1037–1048. Citation on page 48.

KIRNER, E.; SCHUBERT, E.; ZIMEK, A. Good and Bad Neighborhood Approximations for Outlier Detection Ensembles. In: **International Conference on Similarity Search and Applications**. Munich, Germany: Springer, 2017. p. 173–187. Citations on pages 26 and 47.

KNORR, E. M.; NG, R. T. Algorithms for Mining Distance-Based Outliers in Large Datasets. In: **VLDB Conference**. New York, NY, USA: ACM, 1998. p. 392–403. Citations on pages 26, 37, 45, and 67.

KOUDAS, N.; SEVCIK, K. C. High-dimensional Similarity Joins: Algorithms and Performance Evaluation. **Transactions on Knowledge and Data Engineering**, IEEE, v. 12, n. 1, p. 3–18, 2000. Citation on page 42.

KRIEGEL, H.-p.; KRÖGER, P.; SCHUBERT, E.; ZIMEK, A. LoOP: Local Outlier Probabilities. In: **Conference on Information and Knowledge Management**. Hong Kong: ACM, 2009. p. 1649. Citation on page 47.

KRIEGEL, H.-P.; KROGER, P.; SCHUBERT, E.; ZIMEK, A. Interpreting and Unifying Outlier Scores. In: **International Conference on Data Mining**. Mesa, Arizona, USA: SIAM, 2011. p. 12. Citation on page 48.

KRIEGEL, H.-P.; KRÖGER, P.; ZIMEK, A. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering and correlation clustering. **Transactions on Knowledge Discovery from Data**, ACM, v. 3, n. 1, p. 1–58, 2009. Citation on page 47.

KRIEGEL, H.-P.; SCHUBERT, M.; ZIMEK, A. Angle-based Outlier Detection in High-Dimensional Data. In: **International Conference on Knowledge Discovery and Data Mining**. Las Vegas, Nevada, USA: ACM SIGKDD, 2008. p. 444–452. Citations on pages 35 and 47.

LEE, D.-H.; PARK, D.-J. An Efficient Incremental Nearest Neighbor Algorithm for Processing k-Nearest Neighbor Queries with Visal and Semantic Predicates in Multimedia Information Retrieval System. In: **Asia Conference on Asia Information Retrieval Technology**. Jeju Island, Korea: Springer, 2005. p. 653–658. Citation on page 46.

LEE, H.; NG, R. T.; SHIM, K. Similarity Join Size Estimation using Locality Sensitive Hashing. In: **VLDB Conference**. Seattle, Washington, USA: ACM, 2011. v. 4, n. 6, p. 338–349. Citation on page 40.

LESOT, M.-J.; RIFQI, M.; BENHADDA, H. Similarity Measures for Binary and Numerical Data: A Survey. **International Journal of Knowledge Engineering and Soft Data Paradigms**, Inderscience, v. 1, n. 1, p. 63–84, 2009. Citations on pages 30 and 38.

LEYS, C.; KLEIN, O.; DOMINICY, Y.; LEY, C. Detecting Multivariate Outliers: Use a Robust Variant of the Mahalanobis Distance. **Journal of Experimental Social Psychology**, Elsevier, v. 74, p. 150–156, 2018. Citation on page 33.

LI, G.; HE, J.; DENG, D.; LI, J. Efficient Similarity Join and Search on Multi-Attribute Data. In: **International Conference on Management of Data**. Melbourne, Victoria, Australia: ACM SIGMOD, 2015. p. 1137–1151. Citation on page 38.

LI, S.; LEE, R.; LANG, S. D. Mining Distance-based Outliers from Categorical Data. In: **International Conference on Data Mining**. Las Vegas, Nevada, USA: IEEE, 2007. p. 225–230. Citation on page 37.

LI, S.; SHAO, M.; FU, Y. Multi-View Low-Rank Analysis with Applications to Outlier Detection. **Transactions on Knowledge Discovery from Data**, ACM, v. 12, n. 3, p. 22, 2018. Citation on page 41.

LIEBERMAN, M. D.; SANKARANARAYANAN, J.; SAMET, H. A Fast Similarity Join Algorithm Using Graphics Processing Units. In: **International Conference on Data Engineering**. Cancun, Mexico: IEEE, 2008. p. 1111–1120. Citation on page 42.

LIN, Y. S.; JIANG, J. Y.; LEE, S. J. A Similarity Measure for Text Classification and Clustering. **Transactions on Knowledge and Data Engineering**, IEEE, v. 26, n. 7, p. 1575–1590, 2014. Citation on page 38.

LLOYD, S. P. Least Squares Quantization in PCM. **Transactions on Information Theory**, IEEE, v. 28, n. 2, p. 129–137, 1982. Citation on page 34.

LO, M.-L.; RAVISHANKAR, C. V. Spatial Hash-Joins. In: **International Conference on Management of Data**. Montreal, Québec, Canada: ACM SIGMOD, 1996. v. 25, n. 2, p. 247–258. Citation on page 41.

LU, W.; SHEN, Y.; CHEN, S.; OOI, B. C. Efficient Processing of k-Nearest Neighbor Joins using MapReduce. In: **VLDB Conference**. Istanbul, Turkey: ACM, 2012. v. 5, n. 10, p. 1016–1027. Citation on page 49.

LUO, C.; SHRIVASTAVA, A. Arrays of (locality-sensitive) Count Estimators (ACE): Anomaly Detection on the Edge. In: **International World Wide Web Conferences Steering Committee**. Lyon, France: WWW, 2018. p. 1439–1448. Citation on page 48.

LUO, W.; TAN, H.; MAO, H.; NI, L. M. Efficient Similarity Joins on Massive High-Dimensional Datasets Using MapReduce. In: **International Conference on Mobile Data Management**. Bengaluru, India: IEEE, 2012. p. 1–10. Citation on page 49.

MA, J.; PERKINS, S. Time-series Novelty Detection Using One-class Support Vector Machines. In: **International Joint Conference on Neural Networks**. Portland, Oregon, USA: IEEE, 2004. p. 1741–1745. Citation on page 33.

MANN, W.; AUGSTEN, N.; BOUROS, P. An Empirical Evaluation of Set Similarity Join Techniques. In: **VLDB Conference**. New Delhi, India: ACM, 2016. v. 9, n. 9, p. 636–647. Citation on page 38.

MCCAULEY, S.; MIKKELSEN, J. W.; PAGH, R. Set Similarity Search for Skewed Data. In: **Symposium on Principles of Database Systems**. Houston, Texas, USA: ACM SIGMOD-SIGACT-SIGAI, 2018. p. 63–74. Citation on page 41.

MORALES, G. D. F.; GIONIS, A. Streaming Similarity Self-Join. In: **VLDB Conference**. New Delhi, India: ACM, 2016. v. 9, n. 10, p. 792–803. Citation on page 40.

ORAIR, G. H.; TEIXEIRA, C. H. C.; MEIRA, W.; WANG, Y.; PARTHASARATHY, S. Distance-Based Outlier Detection: Consolidation and Renewed Bearing. In: **VLDB Conference**. Singapore: ACM, 2010. v. 3, n. 1-2, p. 1469–1480. Citations on pages 26 and 47.

ORENSTEIN, J. An Algorithm for Computing the Overlay of k-Dimensional Spaces. In: **Advances in Spatial Databases**. Zurich, Switzerland: Springer, 1991. p. 381–400. Citation on page 42.

PAPADIMITRIOU, S.; KITAGAWA, H.; GIBBONS, P. B.; FALOUTSOS, C. LOCI Fast Outlier Detection. In: **International Conference on Data Engineering**. Bangalore, India: IEEE, 2003. p. 315–326. Citations on pages 37, 45, and 67.

PHAM, N.; PAGH, R. A Near-linear Time Approximation Algorithm for Angle-based Outlier Detection in High-dimensional Data. In: **International Conference on Knowledge Discovery and Data Mining**. Beijing, China: ACM SGIKDD, 2012. p. 877–885. Citations on pages 36 and 47.

PICKANDS, J. Statistical Inference Using Extreme Order Statistics. **The Annals of Statistics**, Institute of Mathematical Statistics, v. 3, n. 1, p. 119–131, 1975. Citation on page 33.

RAMASWAMY, S.; RASTOGI, R.; SHIM, K. Efficient Algorithms for Mining Outliers from Large Data Sets. In: **International Conference on Management of Data**. Dallas, Texas, USA: ACM SIGMOD, 2000. v. 29, n. 2, p. 427–438. Citation on page 46.

RAYANA, S. **ODDS Library**. 2016. Available: <http://odds.cs.stonybrook.edu>. Citations on pages 58 and 77.

SARMA, A. D.; HE, Y.; CHAUDHURI, S. ClusterJoin: A Similarity Joins Framework using Map-Reduce. In: **VLDB Conference**. Hangzhou, China: ACM, 2014. v. 7, n. 12, p. 1059–1070. Citation on page 41.

SATHE, S.; AGGARWAL, C. LODES: Local Density Meets Spectral Outlier Detection. In: **International Conference on Data Mining**. Las Vegas, Nevada, USA: SIAM, 2016. p. 171–179. Citation on page 33.

SCHUBERT, E.; ZIMEK, A.; KRIEGEL, H. P. Local Outlier Detection Reconsidered: A Generalized View on Locality with Applications to Spatial, Video, and Network Outlier Detection. **Data Mining and Knowledge Discovery**, Springer, v. 28, n. 1, p. 190–237, 2014. Citations on pages 37, 47, and 55.

SEIDL, T.; FRIES, S.; BODEN, B. MR-DSJ: Distance-Based Self-Join for Large-Scale Vector Data Analysis with MapReduce. In: **Datenbanksysteme für Business, Technologie und Web**. Magdeburg, Germany: Springer, 2013. p. 37–56. Citation on page 41.

SHAHID, N.; NAQVI, I. H.; QAISAR, S. B. Characteristics and Classification of Outlier Detection Techniques for Wireless Sensor Networks in Harsh Environments: A Survey. **Artificial Intelligence Review**, Springer, v. 43, p. 193–228, 2015. Citation on page 29.

SHIM, K.; SRIKANT, R.; AGRAWAL, R. High-dimensional Similarity Joins. **Transactions on Knowledge and Data Engineering**, IEEE, v. 14, n. 1, p. 156–171, 2002. Citation on page 41.

SILVA, Y. N.; AREF, W. G.; ALI, M. H. The Similarity Join Database Operator. In: **International Conference on Data Engineering**. Bangalore, India: IEEE, 2010. p. 892–903. Citation on page 26.

SILVA, Y. N.; REED, J. M. Exploiting MapReduce-based Similarity Joins. In: **International Conference on Management of Data**. Scottsdale, Arizona, USA: ACM, 2012. p. 693–696. Citation on page 41.

STRIPHAS, T. Algorithmic Culture. **European Journal of Cultural Studies**, SAGE, v. 18, n. 4-5, p. 395–412, 2015. Citation on page 25.

TRIPATHI, D.; LONE, T.; SHARMA, Y.; DWIVEDI, S. Credit Card Fraud Detection using Local Outlier Factor. **International Journal of Pure and Applied Mathematics**, Academic Publications, v. 118, n. 7, p. 229–234, 2018. Citation on page 29.

VRIES, T. D.; CHAWLA, S.; HOULE, M. E. Finding Local Anomalies in Very High-dimensional Space. In: **International Conference on Data Mining**. Sydney, Australia: IEEE, 2010. p. 128–137. Citation on page 48.

WANDELT, S.; STARLINGER, J.; BUX, M.; LESER, U. RCSI: Scalable Similarity Search in Thousand(s) of Genomes. In: **VLDB Conference**. Riva del Garda, Trento, Italy: [s.n.], 2013. v. 6, n. 13, p. 1534–1545. Citation on page 38.

WANDELT, S.; WANG, J.; LESER, U.; DENG, D.; GERDJIKOV, S.; MISHRA, S.; MITANKIN, P.; PATIL, M.; SIRAGUSA, E.; TISKIN, A.; WANG, W. State-of-the-art in string similarity search and join. In: **International Conference on Management of Data**. Snowbird, Utah, USA: ACM, 2014. v. 43, n. 1, p. 64–76. Citation on page 49.

WANG, G.; XIAO, C.; LIN, X.; WANG, W.; YU, J. X. Efficient Similarity Joins for near-Duplicate Detection. **Transactions on Database Systems**, ACM, v. 36, n. 3, p. 41, 2011. Citations on pages 38 and 49.

WEI, L.; QIAN, W.; ZHOU, A.; JIN, W. HOT: Hypergraph-based Outlier Test for Categorical Data. In: **Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining**. Seoul, South Korea: Springer, 2003. p. 399–410. Citation on page 37.

WU, S.; WANG, S. Information-theoretic Outlier Detection for Large-scale Categorical Data. **Transactions on Knowledge and Data Engineering**, IEEE, v. 25, n. 3, p. 589–602, 2013. Citation on page 33.

XIA, C.; LU, H.; OOI, B. C.; HU, J. GORDER: An Efficient Method for KNN Join Processing. In: **VLDB Conference**. Toronto, Canada: ACM, 2004. p. 756–767. Citations on pages 40, 42, and 49.

YU, M.; LI, G.; DENG, D.; FENG, J. String similarity search and join: a survey. **Frontiers of Computer Science**, Springer, v. 10, n. 3, p. 399–417, 2016. Citation on page 40.

YU, T.; WANG, X.; SHAMI, A. Recursive Principal Component Analysis-Based Data Outlier Detection and Sensor Data Aggregation in IoT Systems. **Internet of Things Journal**, IEEE, v. 4, n. 6, p. 2207–2216, 2017. Citation on page 48.

ZHANG, K.; HUTTER, M.; JIN, H. A New Local Distance-based Outlier Detection Approach for Scattered Real-world Data. In: **Pacific-Asia Conference on Knowledge Discovery and Data Mining**. Bangkok, Thailand: Springer, 2009. p. 813–822. Citation on page 47.

ZHAO, Y.; NASRULLAH, Z.; LI, Z. PyOD: A Python Toolbox for Scalable Outlier Detection. **arXiv preprint**, 2019. Available: <https://arxiv.org/abs/1901.01588>. Citation on page 48.

ZIMEK, A.; SCHUBERT, E.; KRIEGEL, H. P. A Survey on Unsupervised Outlier Detection in High-Dimensional Numerical Data. **Statistical Analysis and Data Mining**, Wiley, v. 5, n. 5, p. 363–387, 2012. Citations on pages 47 and 48.

APPENDIX

A

BEST PARAMETER VALUES FOR ODSSJ

Dataset	ODSSJ
parkinson	$\varepsilon = 0.7482,\ \tau = 1,\ t = 10$
hepatitis	$\varepsilon = 0.5190,\ \tau = 1,\ t = 10$
glass	$\varepsilon = 0.0758,\ \tau = 4,\ t = 10$
ecoli	$\varepsilon = 0.1869,\ \tau = 7,\ t = 10$
ionosphere	$\varepsilon = 0.6266,\ \tau = 2,\ t = 90$
breastw	$\varepsilon = 0.2346,\ \tau = 7,\ t = 10$
pima	$\varepsilon = 0.5410,\ \tau = 54,\ t = 10$
thyroid	$\varepsilon = 0.0001,\ \tau = 1,\ t = 10$
satimage2	$\varepsilon = 0.7657,\ \tau = 34,\ t = 100$
mammography	$\varepsilon = 0.0130,\ \tau = 65,\ t = 100$
shuttle	$\varepsilon = 0.1266,\ \tau = 1230,\ t = 100$
http	$\varepsilon = 0.2270,\ \tau = 1300,\ t = 100$

Table 8 – ODSSJ 's best parameters found for each dataset.

APPENDIX

B

BEST PARAMETER VALUES FOR HYSORTOD

Dataset	∗HySortOD	HySortOD (Best)
parkinson	$b = 5$	$b = 3$
hepatitis	$b = 5$	$b = 2$
glass	$b = 5$	$b = 28$
ecoli	$b = 5$	$b = 4$
ionosphere	$b = 5$	$b = 4$
breastw	$b = 5$	$b = 6$
pima	$b = 5$	$b = 7$
thyroid	$b = 5$	$b = 69$
satimage2	$b = 5$	$b = 3$
mammography	$b = 5$	$b = 39$
shuttle	$b = 5$	$b = 11$
http	$b = 5$	$b = 8$

∗HySortOD use a fixed parameter value by default.

Table 9 – HySortOD 's best parameters found for each dataset.

APPENDIX

C

BEST PARAMETER VALUES FOR THE STATE-OF-THE-ART ALGORITHMS

Dataset	kNN-Out	DB-Out	LOF	ODIN	∗HilOut	∗aLOCI	∗∗ABOD
parkinson	$k = 4$	$d = 20$	$k = 6$	$k = 2$	$k = 2, h = 11$	$n = 1, g = 1$	-
hepatitis	$k = 11$	$d = 29$	$k = 10$	$k = 9$	$k = 2, h = 10$	$n = 1, g = 1$	-
glass	$k = 1$	$d = 1$	$k = 2$	$k = 12$	$k = 10, h = 9$	$n = 120, g = 2$	-
ecoli	$k = 50$	$d = 1$	$k = 1$	$k = 200$	$k = 100, h = 31$	$n = 50, g = 1$	-
ionosphere	$k = 4$	$d = 2$	$k = 6$	$k = 16$	$k = 7, h = 30$	$n = 1, g = 1$	-
breastw	$k = 2$	$d = 6$	$k = 1$	$k = 2$	$k = 10, h = 22$	$n = 3, g = 4$	-
pima	$k = 413$	$d = 134$	$k = 408$	$k = 219$	$k = 450, h = 19$	$n = 70, g = 1$	-
thyroid	$k = 3$	$d = 30$	$k = 100$	$k = 30$	$k = 9, h = 22$	$n = 17, g = 1$	-
satimage2	$k = 30$	$d = 100$	$k = 150$	$k = 1200$	$k = 64, h = 2$	$n = 1, g = 1$	-
mammography	$k = 1590$	$d = 3$	$k = 185$	$k = 1750$	$k = 12, h = 3$	$n = 3, g = 1$	-
shuttle	$k = 2600$	$d = 2500$	$k = 3400$	$k = 10000$	$k = 7, h = 2$	$n = 1700, g = 1$	-
http	$k = 3000$	$d = 2600$	$k = 2300$	$k = 5000$	-	-	-

∗Parameter values for HilOut and aLOCI are missing because they exceeded the main memory capacity.
∗∗ABOD is a parameter-free algorithm.

Table 10 – Related work's best parameters found for each dataset.


ICMC
USP
SÃO CARLOS