BORDER: Efficient Computation of Boundary Points

Chenyi Xia Wynne Hsu Mong Li Lee Beng Chin Ooi

Abstract

In this work, we investigate the problem of finding boundary points in multi-dimensional datasets. Boundary points are data points that are located at the margin of densely distributed data (e.g. a cluster). In this paper, we propose a simple yet novel approach BORDER (a BOundaRy points DEtectoR) to detect such points. BORDER employs the state-of-the-art database technique - the Gorder kNN join and makes use of the special property of the reverse k-nearest neighbor (RkNN). Our experimental study shows that BORDER detects boundary points effectively and efficiently on various datasets.

I. Introduction

Advancements in information technologies have led to the continual collection and rapid accumulation of data in repositories. Knowledge discovery in databases is a non-trivial process of identifying valid, interesting and potentially valuable patterns in data [13]. Given the urgent need for efficient and effective analysis tools to discover information from these data, many techniques have been developed for knowledge discovery in databases to identify valid, interesting and potentially valuable patterns from the data. Such techniques include data classification and mining association rule, cluster and outlier analysis [15] as well as data cleaning and data preparation techniques to enhance the validity of the data by removing anomalies and artifacts.

In this paper, we examine the problem of *boundary point* detection. Boundary points are data points that are located at the margin of densely distributed data (or cluster). Boundary points are useful in data mining applications since they represent a subset of population that possibly straddles two or more classes. For example, this set of points may denote a subset of population that should have developed certain diseases, but somehow they do not. Special attention is certainly warranted for this set of people since they may reveal some interesting characteristics of the disease. The knowledge of these points is also useful for data mining tasks such as classification [18] since these points can be potentially mis-classified.

Intuitively, boundary points can be defined as follows:

Definition I.1 *A boundary point p is an object that satisfies the following conditions*
i) It is within a dense region $\mathbb{R}$;
ii) $\exists$ region $\mathbb{R}'$ near p, $Den_{\mathbb{R}'} \gg Den_{\mathbb{R}}$ or $Den_{\mathbb{R}'} \ll Den_{\mathbb{R}}$.

Note that *boundary points* are different from outliers [1], [9], [2] or its statistical counterpart

- the change-point [24], [10], [3]. While outliers are located in the sparsely-populated areas, *boundary points* occurs at the margin of dense regions.

We develop a method called BORDER (a BOundaRy points DEtectoR) that utilizes the special property of the reverse k-nearest neighbor (RkNN) [22], and employs the state-of-the-art database technique - the Gorder kNN join [30] to find boundary points in a dataset.

The reverse k-nearest neighbors (RkNN) of an object p are points that look upon p as one of their k-nearest neighbors. A property of reverse k-nearest neighbor is that it examines the neighborhood of an object with the view of the entire dataset instead of the object itself. Hence, it can capture the distribution property of the underlying data and allow the identification of boundary points that lie between two or more distributions.

Utilizing RkNN in data mining tasks will require the execution of a RkNN query for each point in the dataset (the set-oriented RkNN query). However, this is very expensive and the complexity will be $O(N^3)$ since the complexity of a single RkNN query is $O(N^2)$ time using sequential scan for non-indexed data [29], where N is the cardinality of the dataset. In the case where the data is indexed by some hierarchical index structure [5], the complexity can be reduced to $O(N^2 \cdot logN)$. However, the performance of these index structures is often worse than sequential scan in high-dimensional space.

Instead of running multiple RkNN queries, the proposed BORDER approach utilizes Gorder kNN join [30] (or the G-ordering kNN join method) to find the reverse k-nearest neighbors of a set of data points. Gorder is a block nested loop join method that exploits sorting, join scheduling and distance computation filtering and reduction to reduce both I/O and CPU costs. It sorts input datasets into the *G-order* and applies the *scheduled block nested loop join* on the G-ordered data. It also employs distance computation reduction to further lower the CPU costs. It doesn't require an index and handles high-dimensional data efficiently.

BORDER processes a dataset in three steps. First, it executes Gorder kNN join to find the k-nearest neighbors for each point in the dataset. Second, it counts the *number of reverse k-nearest neighbors* (RkNN number) for each point according to the kNN-file produced in the first step. Third, it sorts the data points according to their RkNN number and finally, the boundary points whose RkNN number is smaller than a user predefined threshold can be incrementally output. Our experimental study shows that the proposed BORDER method is able to detect boundary points effectively and efficiently. Moreover, it helps the density-based clustering method DBScan [12] to find out the correct clusters and improves the classification accuracy for various classifiers. Yet we shall elucidate that BORDER is based on the observation that boundary points tend to have fewer reverse k-nearest neighbors. This assumption is usually true when the dataset contains well-clustered data. However, for some real-world dataset such that the data are not well-clustered and the boundary is not so clear, this assumption may be invalidated and BORDER would fail to find correct boundary points.

The remainder of the paper is organized as follows. Section 2 introduces RkNN and the kNN join. Section 3 describes BORDER in detail. Section 4 presents the results of our performance study. Section 5 reviews related work. Finally, Section 6 concludes the paper.

II. Preliminary

In this section, we introduce the concepts of the reverse k-nearest neighbor and the k-nearest neighbor (kNN) join.

A. Reverse k-Nearest Neighbor

The reverse k-Nearest neighbor (RkNN) has been proposed in [22] and has received considerable attention in the recent years. Recent work [22], [28], [31], [27], [21] have highlighted the importance of reverse nearest neighbor (RNN) queries in decision support systems, profile-based marketing, document repositories and management of mobile devices. RkNN is formally defined as follows.

Definition II.1 (Reverse k-Nearest Neighbor) *Given a dataset DB, a query point p, a positive integer k and a distance metric $\ell(\ )$, reverse k-nearest neighbors of p, denoted as $RkNN_p(k)$ is a set of points p_i that $p_i \in DB$ and $\forall p_i, p \in kNN_{p_i}(k)$, where $kNN_{p_i}(k)$ are the k-nearest neighbors of point p_i.*

RkNN has properties that are uniquely different from the conventional k-nearest neighbors (kNN).

1) Reverse k-nearest neighbors are not localized to the neighborhood of the query point.
2) The cardinality of a point's reverse k-nearest neighbors varies by data distribution.

Figure 1 gives an example of RkNN. Suppose p_2 is the query point and k=2. From the diagram, we see that p_2 is one of the 2-nearest neighbors of p_1, p_3, and p_4. Hence, the reverse 2-nearest

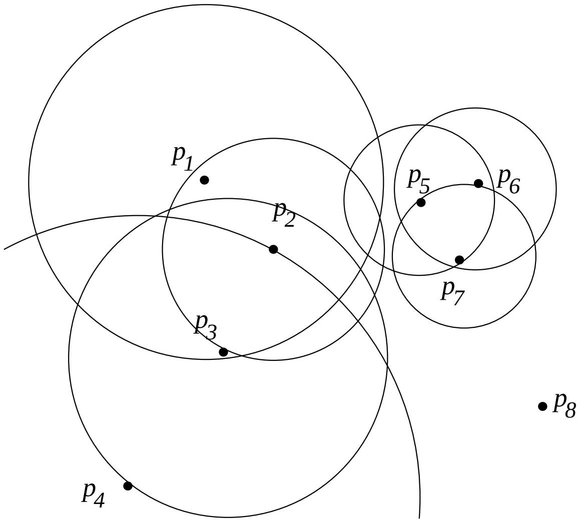


Fig. 1. Example of an RkNN Query.

neighbors of p_2 are p_1, p_3, and p_4. Note that although p_4 is far from the query point p_2, it is still an R2NN of p_2. In contrast, p_5 and p_7 are close to p_2 but they are not answers of the R2NN query of p_2. Moreover, p_2 has 3 reverse 2-nearest neighbors while p_4 and p_8 has 0 reverse 2-nearest neighbor.

These properties of RkNN have potential applications in the area of data mining. However, the complexity of RkNN remains a bottleneck. The naive solution for RkNN search first computes k-nearest neighbors for each point p in the dataset and thereafter, retrieves points that have q as one of the k-nearest neighbors and outputs them as answers. The complexity of this naive solution is $O(NlogN)$ for data indexed by some hierarchical structure such as the R-tree[14] or $O(N^2)$ for non-indexed data, where N is the cardinality of the dataset.

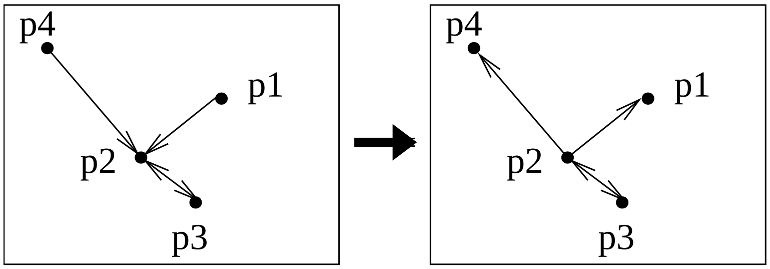
Methods for processing RkNN queries utilize the geometry properties of RkNN to find a small number of data points as candidates and then verify them with nearest neighbor queries or range queries [25], [29], [27]. However, these techniques are not scalable with data dimensionality and the value k. In addition, data analysis utilizing RkNN usually runs a set of RkNN queries for all points in a dataset. Therefore, methods for a single RkNN query are not efficient for set-oriented RkNN queries.

In contrast, the proposed BORDER approach will utilize an efficient operation called kNN-join *k-nearest neighbor join* to compute RkNN queries.

B. kNN Join

The kNN-join [30], [6], [7] combines each point in a dataset R with its k-nearest neighbors in another dataset S. R is called the outer dataset (or the *query* dataset where the all points in R are the query points), and S is called the inner dataset. When the outer and inner datasets are the same, the kNN-join is also known as the *kNN self-join* [6]. With its set-a-time nature, kNN-join can be used to efficiently support various applications where a large number kNN queries are involved.

Definition II.2 (kNN-join) *Given two data sets R and S, an integer K and the similarity metric $dist()$, the kNN-join of R and S, denoted as $R \ltimes_{kNN} S$, returns pairs of points (p_i, q_j) such that p_i is from the outer dataset R and q_j from the inner dataset S, and q_j is one of the k-nearest neighbors of p_i.*



(a) kNN graph (b) RkNN graph

Fig. 2. kNN vs. RkNN

The kNN join can be used to answer the set-oriented RkNN query (the RkNN join) given the following relationship that is inherent between the kNN join and the RkNN join.

Lemma II.1 *The reverse k-nearest neighbors of all points in dataset DB can be derived from the k-nearest neighbors of all points in DB. By reversing all pairs (p_i, p_j) produced by kNN-join, we obtain the complete set of pairs (p_j, p_i) that p_i is p_j's reverse k-nearest neighbor.*

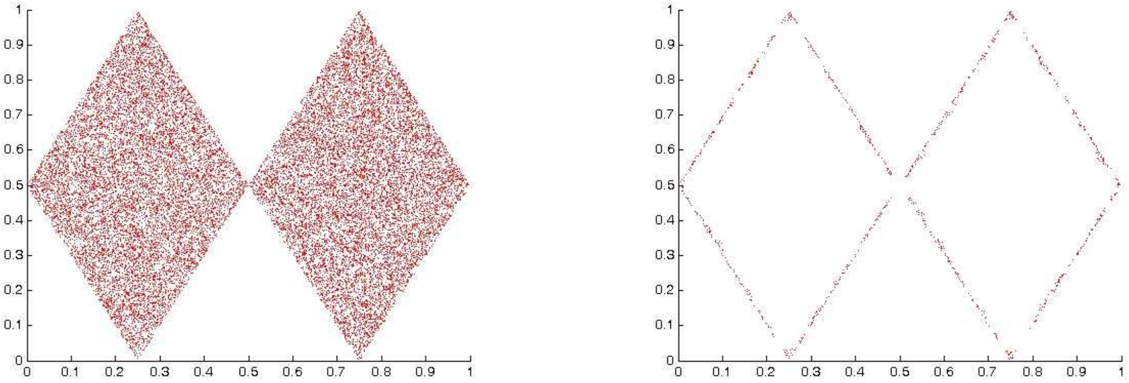
Proof: Figure 2 illustrates the kNN and RkNN relationship with an edge $\overrightarrow{p_ip_j}$. Figure 2(a) is the kNN graph and each edge $\overrightarrow{p_ip_j}$ denotes a kNN pair (p_i, p_j) such that p_j is p_i's kNN. Figure 2(b) is the RkNN graph and each edge $\overrightarrow{p_ip_j}$ denotes a RkNN pair (p_i, p_j) such that p_j is p_i's RkNN. Given the kNN of all points in a dataset, we can derive the RkNN of each point by simply reversing the direction of the edges in the kNN graph. Hence, we have the lemma.

Therefore, in BORDER, we utilize the kNN join to process the set oriented RkNN queries efficiently.

III. BORDER

In this section, we describe the details of BORDER, a method for efficient boundary point detection. The basic idea of BORDER is based on the observation that boundary points tend to have fewer reverse k-nearest neighbors.

Figure 3 shows the results of our preliminary study. Given a 2-dimensional dataset as shown in Figure 3(a), we plot the points whose reverse 50-nearest neighbors answer set contain less than 30 points. Figure 3(b) shows that the boundaries of the clusters are clearly defined by those points having fewer number of RkNN.



(a)

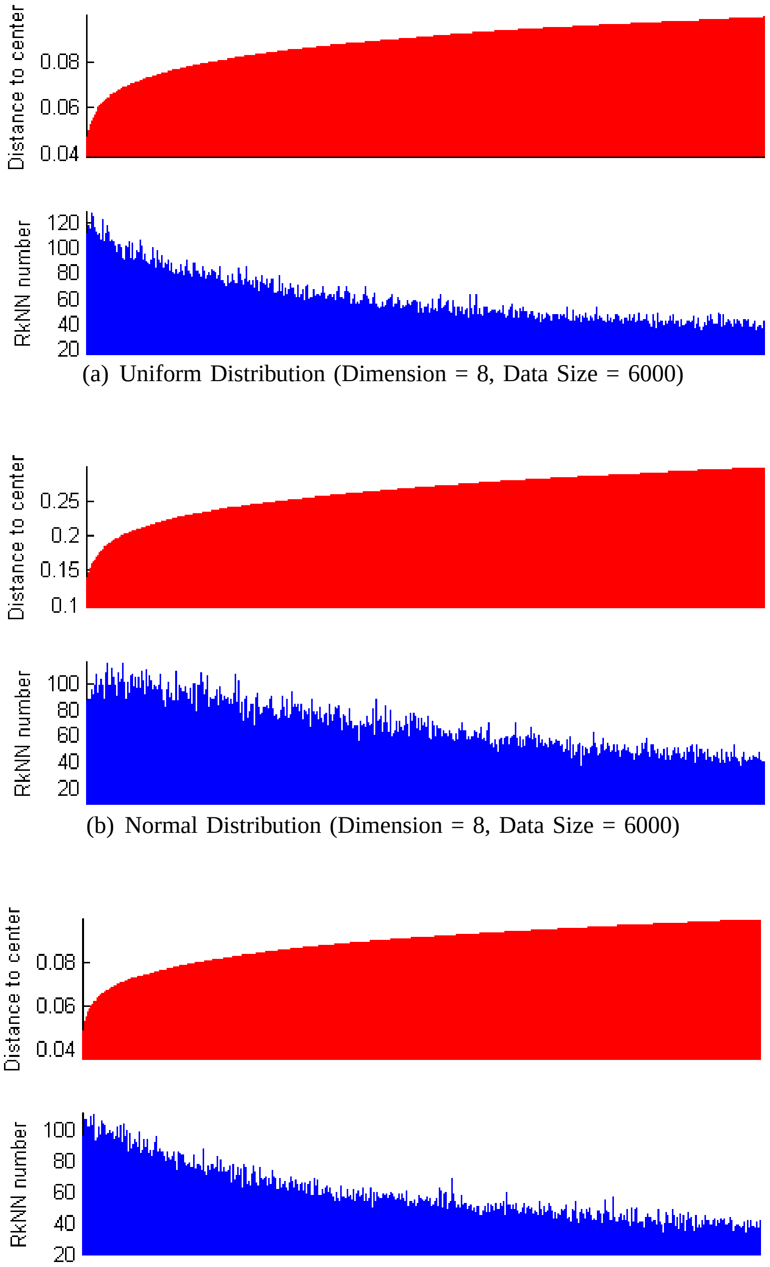
(b)

Fig. 3. Preliminary Study I.

We also carry out another preliminary study to find out the relationship between the location of a point p and the number of its RkNN in high-dimensional spaces.

In order to determine the boundary of a densely distributed region, we use hyper-sphere datasets [1] which contain the dense regions of the shape of the high-dimensional spheres. This is because the boundary points of spherical regions are always located at the area farthest from the center of the sphere.

[1] The generation of hyper-sphere data is given in the experiment section.



(a) Uniform Distribution (Dimension = 8, Data Size = 6000)

(b) Normal Distribution (Dimension = 8, Data Size = 6000)

(c) Zipf Distribution (Dimension = 8, Data Size = 6000)

Fig. 4. Preliminary Studies.

Figure 4 summarizes the results of the experiments on the hyper-sphere datasets of different distributions. We compute the number of reverse k-nearest neighbors of each point in the dataset and the distance of each point to the center of the cluster that the point belongs to. We then sort the data points according to the distance of each point to the center of the cluster that the point

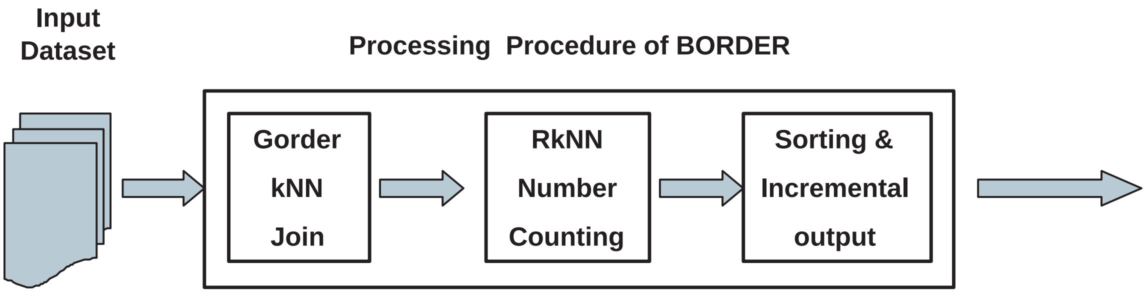


Fig. 5. Overview of BORDER

belongs to and plot the distance to cluster center and the number of reverse k-nearest neighbors of each point as in Figure 4. Each vertical line in the graphs in Figure 4 is corresponding to one data point. The height of the lines in the upper sub-graphs represents the distance to cluster center and the height of the lines in the lower sub-graphs is corresponding to the number of reverse k-nearest neighbors of each point. The study shows clearly that the number of RkNN decreases as the distance of a point from the center increases. This result confirms that for well-clustered datasets in high-dimensional spaces, the boundary points which lie at the margin of the clusters tend to have fewer reverse k-nearest neighbors.

Figure 5 gives an overview of BORDER. It comprises of three main steps:

1) A kNN-join operation with Gorder to find the k-nearest neighbors for each point in the dataset.
2) An RkNN counter to obtain each point's RkNN number (the cardinality of each point's RkNN answer set).
3) Points are sorted according to their RkNN number. Points that its RkNN number is smaller than a user defined threshold are output incrementally as boundary points.

In the following sections, we will give the details of each step.

A. kNN Join

The core of BORDER is the efficient kNN join algorithm - Gorder [30], which is an optimized block nested loop join with efficient data scheduling and distance computation filtering.

Algorithm 1 presents the kNN self-join with Gorder. It has two phases.

1) **G-ordering** (line 1)

Algorithm 1 Gorder(R)

Input:

R is a dataset.

Output:

kNN-file

Description:

```
 1: **G_Ordering**($R$);
 2: **for each** block $B_r \in R$ **do**
 3:    ReadBlock($B_r$);
 4:    Sort Blocks of $B'_r$ in $R$ according to MinDist($B'_r, B_r$);
 5:    **for each** $B'_r \in R$ **and** MinDist($B_r, B'_r$)$\leq$ PrunDist($B_r$) **do**
 6:       ReadBlock($B'_r$);
 7:       Divide $B_r$ and $B'_r$ into sub-blocks $b_r$ and $b'_r$;
 8:       **for each** sub-block $b_r \in B_r$ **do**
 9:          SortBlocks($b_r, b'_r$);
10:          **for each** sub-block $b'_r \in B'_r$ **and** MinDist($b_r, b'_r$)$\leq$ PrunDist($b_r$) **do**
11:             **for each** point $p_r \in b_r$ **and** MinDist($b_r, b'_r$) $\leq$ PrunDist($p_r$) **do**
12:                **for each** point $p'_r \in b'_r$, ComputeDist($p_r, p'_r$);
13:    OutputkNN(kNN-file);
```

2) **Scheduled block nested loop join** (line 2-13)

The *G-ordering* performs a PCA (principal component analysis) transformation of R and then sorts R into the *Grid Order*. The *Grid Order* is defined as below [30].

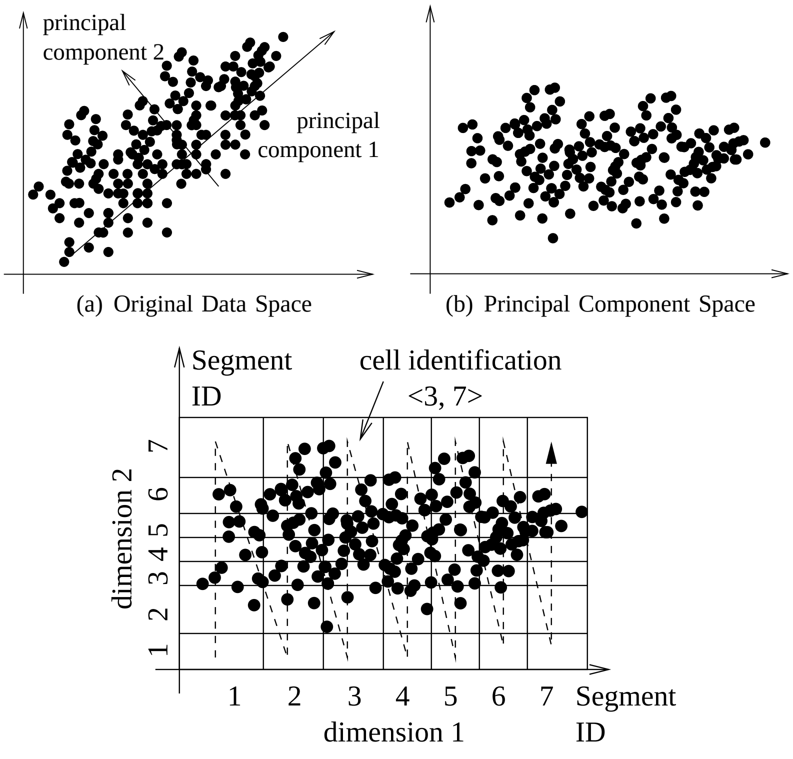
Definition III.1 ***(Grid order*** $\prec_g$ ***)*** *Given a grid which partitions the d-dimensional data space into l^d rectangular cells, points $p_m \prec_g p_n$ if and only $\nu_m \prec \nu_n$, where l is the number of segments per dimension and ν_m (or ν_n) is the* identification vector *of the cell surrounding point p_m (or p_n). $\nu = < s_1, ..., s_d >$, where s_i is the segment number to which the cell belongs on the ith dimension.*

$\nu_m \prec \nu_n$ if and only if a dimension k exists that, $\nu_m.s_k < \nu_n.s_k$ and $\nu_m.s_j = \nu_n.s_j$, for $\forall j < k$.

Essentially, the grid order sorts the data points according to the cell surrounding the point lexicographically as illustrated in Figure 6.

The G-ordered data exhibit two interesting **properties**:

1) Most of the information in the original space is condensed into the first few dimensions along which the variances in the data distribution are the largest [11].
2) Given two blocks of G-ordered data B and B', minimum distance between B and B'



(a) Original Data Space

(b) Principal Component Space

(c) Grid Order

Fig. 6. Illustration of G-ordering.

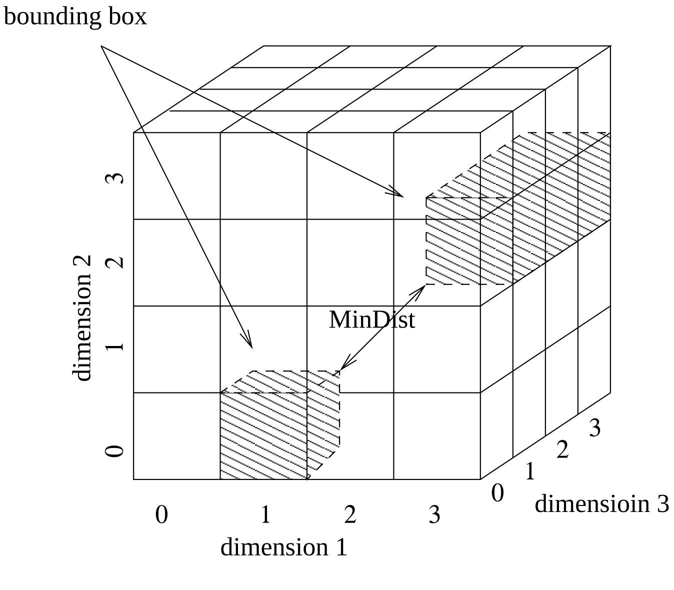


Fig. 7. Illustration of MinDist.

MinDist(B,B') can be calculated according to the *bounding box* of B and B' (see Figure 7). The *bounding box* of B and B' can be computed by examining the first point p_1 and last point p_m of the G-ordered data as described in [30].

These properties of the G-ordered data are used in Gorder for join scheduling and distance computation reduction.

Next, let us examine the *scheduled block nested loop join* in lines 2-13 of Algorithm 1. The join stage of Gorder employs the *two-tier partitioning strategy* to optimize the I/O time and CPU time separately.

1) First tier (line 2-6): The first tier of Gorder partitions the G-ordered input dataset into blocks consisting of several physical pages. Blocks of R B_r are loaded into memory sequentially. We call B_r the *query block* as all points is B_r are regarded as query points. For each B_r in memory, blocks of R B'_r are sorted in the ascending order of the MinDist between B_r and B'_r. Each block B'_r that cannot be pruned by the *pruning distance* of B_r are loaded into memory to join with B_r in the second tier processing.

   The *pruning distance* of B_r is the maximal **kNN-distance** (the distance between a point and its kth-nearest neighbor) of all points in it. After all blocks of R are either joined with B_r or pruned, the kNN of all points in B_r are output into a kNN-file which records the each kNN pairs (line 13).

2) Second tier (line 7-12): The second tier processing of Gorder joins two blocks B_r and B'_r in memory in the similar way as the first tier. The blocks in memory are divided into sub-blocks. For each sub-block b_r, the sub-blocks b'_r in B'_r are sorted according to their minimum distance to b_r. Those unpruned sub-blocks b'_r participate in the join with sub-block b_r one by one.

   To join two sub-block b_r and b'_r, for each point p_r in b_r, we examine whether MinDist(b_r, b'_r) is greater than the pruning distance of p_r. The pruning distance of p_r is its kNN-distance. If true, b'_r cannot contain any points that are k-nearest neighbors of p_r and so can be skipped. Otherwise, function *Compute_Dist* is called to compute the distance between p_r and each data point p'_r in b'_r. *Compute_Dist* employs distance computation reduction technique utilizing property 1 of the G-ordered data to reduce CPU time [30].

At the end of Gorder, the k-nearest neighbors of all points in R are found and saved in the

Algorithm 2 RkNN_Counter(R, kNN-file)

Input:

R: the input dataset; kNN-file: a file records k-nearest neighbors of each points in R.

Description:

```
for  each point $p \in R$ do
    Read its k-nearest neighbors $kNN_p(k)$ from kNN-file;
    for each point $p_i \in kNN_p(k)$ do
        increase $rnum_{p_i}$ by 1;
```

Algorithm 3 Sort_and_Output(R, $k_{threshold}$)

Input:

R: the input dataset;

Description:

```
Sort points in R in ascending order according to their RkNN number;
for Points $p_i$ in R do
    if $rnum_{p_i} < k_{threshold}$ then
        Output $p_i$;
```

kNN-file.

B. RkNN Counter

In this step, BORDER counts the number of reverse k-nearest neighbors (RkNN number) for each point p (denoted as $rnum_p$) utilizing the kNN information saved in the kNN-file. According to the reversal-ship between kNN and RkNN which we have discussed in Lemma II.1, the number of each point's k-nearest neighbor can be obtained by a scanning of the kNN-file and for each point p_i in the kNN set of a point p, increasing $rnum_{p_i}$ by 1.

Algorithm 2 depicts the count procedure.

C. Sorting

Data points then can be sorted according to their RkNN number so that they can be output incrementally. We let $k_{threshold}$ be a user defined threshold which is tunable. For all point p, if its RkNN number $rnum_p < k_{threshold}$, they are output as detected boundary points. Algorithm 3 shows the details.

D. Cost Analysis

Next, we analyze the I/O and CPU cost of BORDER.

The major cost of BORDER lies in the kNN join procedure. The number of I/Os incurred during the kNN join procedure in terms of the number of page is [30]:

$$3N_r + 2N_r \left(\left\lceil log_{B-1}\tfrac{N_r}{B}\right\rceil + 1\right) + \tfrac{N_r}{n_r} \cdot N_r \cdot \gamma_1$$

where N_r is the total number of R data pages, n_r is the allocated buffer pages for query data, and B is the total buffer pages available in memory.

The number of page writes is N_{knn} which is incurred in the kNN join procedure to output the k-nearest neighbors. N_{knn} is the total number of pages for the k-nearest neighbors information.

The number of page reads in the RkNN counter procedure is N_{knn} for scanning the kNN file.

Hence, the total I/O time is:

$$\left(3N_r + 2N_r \left(\left\lceil log_{B-1}\frac{N_r}{B}\right\rceil + 1\right) + \frac{N_r}{n_r} \cdot N_r \cdot \gamma_1 + N_{knn}\right) \cdot T_{io\_read} + N_{knn} \cdot T_{io\_write}$$

where T_{io_read} (T_{io_write}) is the time for reading (writing) one page.

The major CPU cost of BORDER is the distance computation in the kNN join phase. The number of distance computations is:

$$P_r^2 \cdot \gamma_2$$

where P_r is the number of objects in the dataset, γ_2 is the selectivity of distance computation.

Hence, the total CPU time is:

$$P_r^2 \cdot \gamma_2 \cdot T_{dist}$$

where T_{dist} is the time for one distance computation.

The selectivity ratio γ_1 and γ_2 are estimated as following [30]:

$$\gamma = \sum_{k=0}^{d} \left(\begin{array}{c} d \\ k \end{array}\right) \left(2 \cdot \sqrt[d]{\frac{M}{P_r}}\right)^k \cdot V_{sphere}^{d-k}(\varepsilon) \tag{1}$$

where

$$V_{sphere}^{d-k}(\varepsilon) = \frac{\sqrt{\pi^{d-k}}}{\Gamma\left(\frac{d-k}{2}+1\right)} \cdot \varepsilon^{d-k} \tag{2}$$

$$\varepsilon = \sqrt[d]{\frac{K \cdot \Gamma(d/2+1)}{P_r}} \cdot \frac{1}{\sqrt{\pi}} \tag{3}$$

$$\Gamma(x+1) = x\Gamma(x), \quad \Gamma(1) = 1, \quad \Gamma(1/2) = \sqrt{\pi} \tag{4}$$

Where M is the block size, that is, the number of data points in one block. If we replace M with the size of block (M_1) in the first tier of the kNN join procedure, we obtain γ_1. And if we replace M with the size of sub-block (M_2) in the second tier of the kNN join procedure, we obtain γ_2.

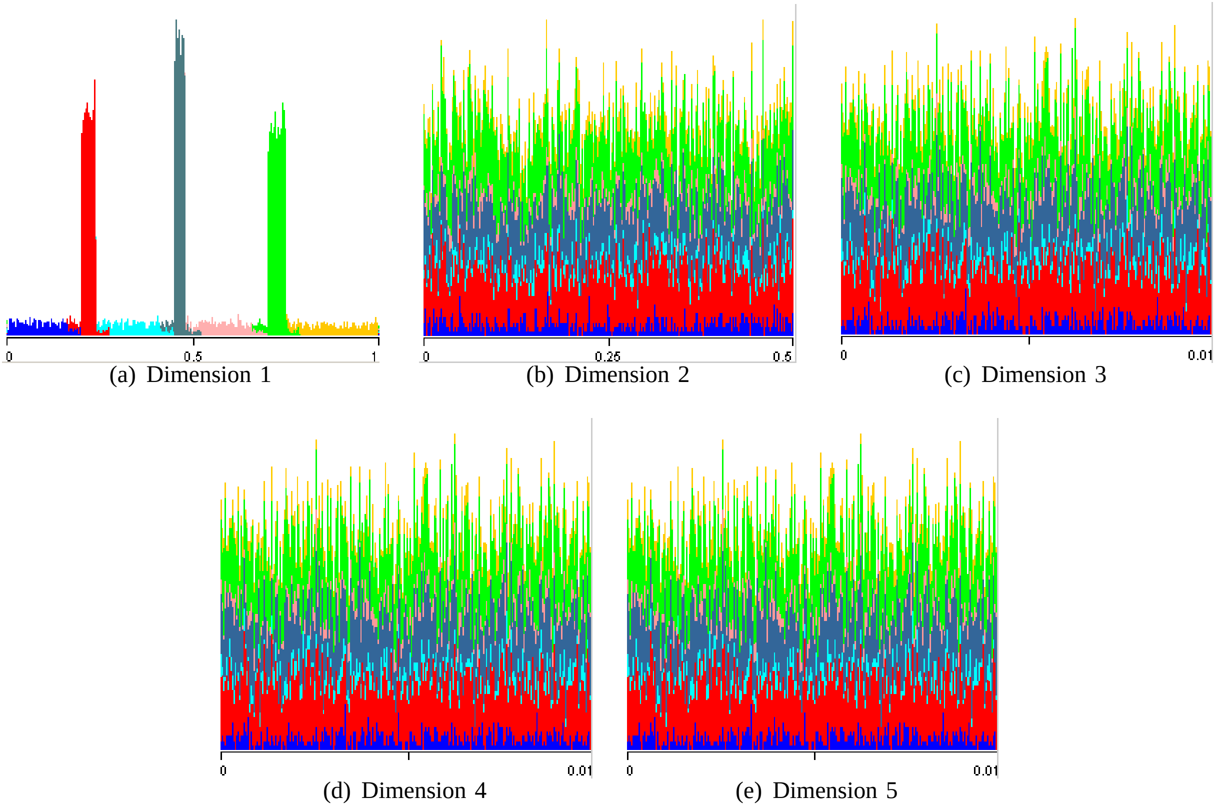
IV. Performance Study

We conducted extensive experimental study to evaluate the performance of BORDER and present the results in this section. We implemented BORDER in C++ and studied its efficiency and effectiveness as follows:

1) Effectiveness: We apply BORDER to 3 types of datasets: I) a set of high-dimensional hyper-sphere datasets with various data distributions and sizes; II) a set of 2-dimensional clustered dataset of arbitrary cluster shapes; III) clustered dataset with mixed clusters IV) the labelled datasets for classification.

   a) Dataset I: The hyper-sphere datasets are used to demonstrate the ability of BORDER in detecting boundary points in high-dimensional spaces. The hyper-sphere datasets are generated as follows: Given the distribution, the number and the centers and the radii of the hyper-spheres, data points are generated according to the specified distribution. Points that are within the defined hyper-spheres are inserted into the dataset, whereas points that are outside of the hyper-spheres are discarded. To show the location of the found boundary points, we present the distribution of the $\ell(p, c)$, where p is a detected point, c is the center of the hyper-sphere which p belongs to, and $\ell(p, c)$ is the distance between p and c.

   b) Dataset II: This set of datasets are to exhibit the ability of BORDER to find out boundary points located at the border of *arbitrary-shaped* clusters *visually*. Therefore, we use the 2-dimensional datasets so that we can plot the detected boundary points in a plane to show the effectiveness of BORDER.



(a) Dimension 1 (b) Dimension 2 (c) Dimension 3

(d) Dimension 4 (e) Dimension 5

Fig. 8. Data distribution of Dataset IV on each dimension.

c) Dataset III: In this dataset, the dense clusters mix with some less dense clusters. In such case, the traditional density-based clustering method (e.g, DBScan) cannot identify the clusters properly. We show that removing the boundary points helps DBScan to find the clusters correctly. The removed boundary points can be inserted back into the identified clusters with a post-processing procedure by checking their connectivity and density.

d) Dataset IV: This synthetic dataset contains 7 classes with 5 attributes. The dataset is generated as the following: We divide the first dimension into 7 segments. Each segment is corresponding to one class. We assign points within each segment to its corresponding class and those points lying at the adjacent region of each segment to different classes. Thus, the 7 classes do not have a distinct separable boundaries. Along other dimensions, data points are distributed randomly. Figure 8 shows the data distribution of the dataset on each dimension. Note that along dimension 1, points

Distribution	Cluster Number	Size	Radius	Dimension	k	Processing Time (Sec)
Normal	1	6000	0.3	8	50	6.547
Normal	5	6000	0.3	10	50	4.313
Zipf	3	6000	0.2	6	50	4.625
Zipf	2	6000	0.5	4	50	2.89
Uniform	2	6000	0.2	8	50	3.938
Uniform	2	6000	0.1	12	50	5.875

TABLE I

HYPER-SPHERE DATASETS AND PROCESSING TIME.

Distribution	Dimension	Radius	Mean	Standard Deviation
Normal	8	0.3	0.2868	0.0141
Normal	10	0.3	0.2868	0.0133
Zipf	6	0.2	0.1955	0.0043
Zipf	4	0.5	0.49	0.0092
Uniform	8	0.2	0.1959	0.0037
Uniform	12	0.1	0.0981	0.0019

TABLE II

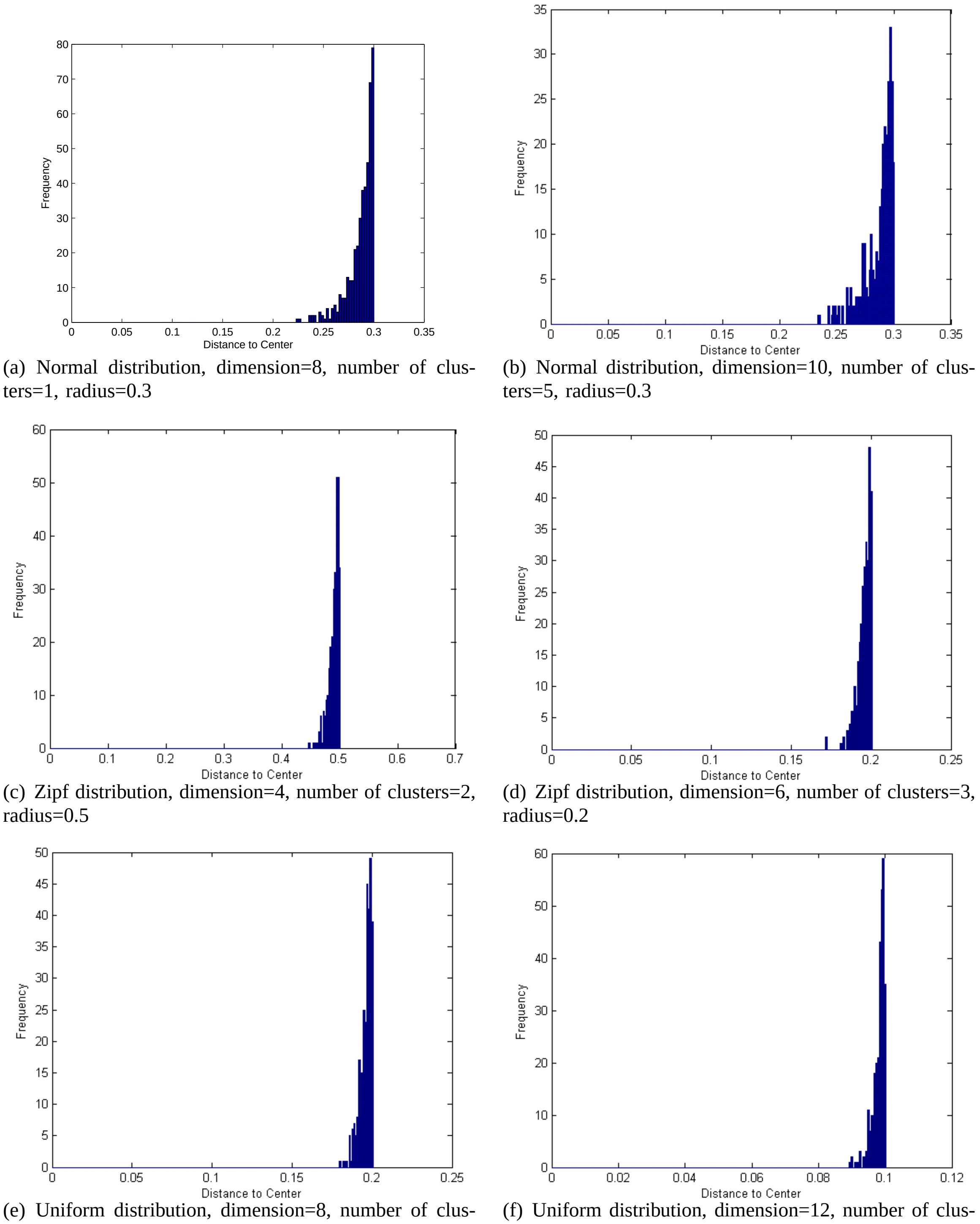
MEAN AND STANDARD DEVIATION OF THE DISTANCE OF DETECTED BOUNDARY POINTS TO THE HYPER-SPHERE CENTER.

that are located at the boundary region of two adjacent classes are belong to different classes. We use this dataset to show as an example that by removing the boundary points which straddle two classes of difference density can improve the accuracy of the classifier.

2) Efficiency: We compare the response time of BORDER using the Gorder [30] with other kNN query methods such as the nested loop join and the MuX (an R-tree like structure which has better performance than the R-tree) [8], [6]. The datasets used here are the synthetic cluster datasets generated using the method described in [19].

A. Evaluation of Effectiveness

1) On Hyper-sphere Datasets: We first study the effectiveness of BORDER on the hyper-sphere datasets of various distributions, different numbers of dimension and containing different



(a) Normal distribution, dimension=8, number of clusters=1, radius=0.3

(b) Normal distribution, dimension=10, number of clusters=5, radius=0.3

(c) Zipf distribution, dimension=4, number of clusters=2, radius=0.5

(d) Zipf distribution, dimension=6, number of clusters=3, radius=0.2

(e) Uniform distribution, dimension=8, number of clusters=2, radius=0.3

(f) Uniform distribution, dimension=12, number of clusters=2, radius=0.1

Fig. 9. Study on hyper-sphere datasets.

	Size	Dimension	k	Processing Time (Sec)
dataset 1	40000	2	50	9.985
dataset 2	10680	2	50	2.781
dataset 3	9050	2	50	2.562
dataset 4	12950	2	50	3.469

TABLE III
DATASETS (WITH CLUSTERS) AND PROCESSING TIME.

number of clusters. Figure 9 summarizes the experiment results. We incrementally output 300 points with lowest RkNN number of as boundary points. For each graph, the x-axis is the point p's distance to the center of the hyper-sphere that p belongs to ($\ell(p, c)$) and the y-axis is the frequency. We observe that for all datasets, $\ell(p, c)$ of the output points by BORDER is equal or close to the radii of the hyper-spheres. Table II summarizes the mean and standard deviation of the distance of detected boundary points to the hyper-sphere center. Both the plotting and statistic information of the output of BORDER indicate those points are indeed boundary points of the hyper-spherical-shaped clusters. The properties and the processing time of BORDER on the hyper-sphere datasets are presented in Table I. BORDER processes them efficiently.

2) On Arbitrary-shaped Clustered Datasets: Next, we study the effectiveness of BORDER on the datasets containing arbitrary-shaped dense regions. We select 2-dimensional datasets in order to visualize the results. BORDER is also applicable to high-dimensional spaces.

This set of experiments are carried on the two dimensional cluster dataset. Figure 10 demonstrates the incremental output of BORDER executed on dataset 1. The thresholds are set as 30, 33 and 35. The graphs in Figure 10 shows that the plotted points outline the boundaries of the clusters in dataset clearly. In addition, note that with the incremental output, we can stop whenever we are satisfied with the quality of detected boundary points. Figure 11 shows the results of boundary point detection on datasets 2, 3, 4. It is clear that BORDER can find the boundary points effectively. The processing time of BORDER is summarized in Table III. It shows that BORDER is very efficient for processing these datasets.

3) On Mixed Clustered Dataset: In the second set of experiments, we mix the dense clusters with less dense clusters and study the ability of using BORDER for preprocessing data for clustering. Figure 12 [12] shows that on such a dataset, it is difficult for DBScan to identify the correct clusters. In Figure 12 (b), (c) and (d), we plot the clusters detected by DBScan with different colors. We observe that if we set the density requirement of DBScan high, that is, Eps=0.00967926, MinPts=10, points in the sparse cluster are all regarded as outliers (Figure 12 (b)). If we set the density requirement of DBScan low, that is, Eps=0.0223533 and MinPts=10 (Figure 12 (c)) or Eps=0.0469042, MinPts=10 (Figure 12 (d)), DBScan returns clusters mixing dense and sparse regions.

Figure 12 (e) illustrates the dataset after we remove the boundary points ($k_{threshold} = 40$). Figure 12 (f) shows the result of DBScan working on the dataset after the boundary points are removed. DBScan with parameters Eps=0.0469042 and MinPts=10 can easily identify the dense clusters as well as the sparse clusters correctly because they are now well separated. The removed the boundary points can be inserted into the clusters with a post-processing procedure which examines the density of the points and their connectivity with the clusters.

4) On the Labelled Dataset for Classification: Finally, we conduct experiments on the labelled dataset for classification. We test various classification methods provided by Weka [17] and compare the classification accuracy before and after we remove some detected boundary points. The test accuracy is evaluated by 10-folders cross validation. The results show that removing the boundary points reduces the ratio of misclassified data points and improves the classification accuracy effectively.

Table IV and Table V summarizes the results when we define different thresholds for the RkNN number. When we set the $k_{threshold}$ 25 or 30, the average improvement ratios in terms of incorrectly classified ratio are 20.03% and 43.51% respectively and the average improvement ratios in terms of incorrectly classified instance are 22.07% and 46.60% respectively.

B. Evaluation of Efficiency

We now study the efficiency of BORDER on the synthetic datasets of high dimensionality and variable sizes. The most expensive step of BORDER lies in the kNN join procedure. In the following, we compare the performance of the Gorder based BORDER with the performance of BORDER using other methods (NLJ and MuX) for kNN computation.

1) Effect of Dimensionality: We first evaluate BORDER on datasets of dimensionality from 8 to 64. Figure 13 presents the results on the 100K clustered datasets. The graph shows that

	Before		After ($k_{threshold}$=25)		improvement	
Classification Method	Incorrectly Classified Ratio	Incorrectly Classified Instances	Incorrectly Classified Ratio	Incorrectly Classified Instances	Incorrectly Classified Ratio	Incorrectly Classified Instances
Decision Table	3.20%	391	2.57%	306	19.69%	21.74%
OneR	3.26%	398	2.21%	263	32.21%	33.92%
Nnge	3.25%	397	2.12%	252	34.77%	36.52%
Jrip	3.43%	418	2.36%	280	31.20%	33.01%
AdaBoost M1	19.14%	2335	18.13%	2156	5.28%	7.67%
MultiBoost AB	19.14%	2335	18.13%	2156	5.28%	7.67%
Raced Incremental Logit Boost	3.20%	391	2.53%	301	20.94%	23.02%
IB1	15.48%	1888	14.02%	1667	9.43%	11.71%
Naive Bayes Simple	3.48%	425	2.57%	306	26.15%	28.00%
SMO	5.93%	723	5.02%	597	15.35%	17.43%
Average	7.95%	970.1	6.97%	828.4	20.03%	22.07%

TABLE IV

COMPARISON OF CLASSIFICATION ACCURACY ($k_{threshold}$=25).

BORDER using NLJ is very expensive and the response time increases quickly when data dimensionality increases. BORDER with Gorder is shown to be the most efficient method and scalable to high-dimensional data. The response time increases moderately while data dimensionality increase. The speed-up factor of BORDER with Gorder over BORDER with MuX increases from 0.68 at dimensionality of 8 to 2.9 at dimensionality of 64. The study demonstrates that Gorder is the best choice for BORDER and works efficiently for high dimensional data.

2) Effect of Data Size: Next, we study the performance of BORDER with varying dataset size. The clustered datasets are in the 16-dimensional space and their sizes vary from 10,000 to 1,000,000 objects. The results are summarized in Figure 14.

We observe that BORDER with NLJ (Nested Loop Join) performs the worst choice for all the datasets. Its response time increases exponentially with the data size. Comparing BORDER with Gorder [30] and MuX [6], we find that Gorder is the more efficient method for datasets of

	Before		After ($k_{threshold}$=30)		improvement	
Classification Method	Incorrectly Classified Ratio	Incorrectly Classified Instances	Incorrectly Classified Ratio	Incorrectly Classified Instances	Incorrectly Classified Ratio	Incorrectly Classified Instances
Decision Table	3.20%	391	1.37%	158	57.19%	59.59%
OneR	3.26%	398	1.34%	155	58.90%	61.06%
Nnge	3.25%	397	1.26%	145	61.23%	63.48%
Jrip	3.43%	418	1.40%	162	59.18%	61.24%
AdaBoost M1	19.14%	2335	16.90%	1950	11.70%	16.49%
MultiBoost AB	19.14%	2335	16.90%	1950	11.70%	16.49%
Raced Incremental Logit Boost	3.20%	391	1.39%	160	56.56%	59.08%
IB1	15.48%	1888	12.44%	1435	19.64%	23.99%
Naive Bayes Simple	3.48%	425	1.36%	157	60.92%	63.06%
SMO	5.93%	723	3.67%	423	38.11%	41.49%
Average	7.95%	970.1	5.80%	669.5	43.51%	46.60%

TABLE V

COMPARISON OF CLASSIFICATION ACCURACY ($k_{threshold}$=30).

variable sizes and the speed-up factor of Gorder over MuX ranges from 0.51 to 2.6. The study shows that BORDER with Gorder kNN join is scalable to large data size.

V. RELATED WORK

BORDER is the first work proposed for boundary points detecting. It utilizes advanced database operation kNN-join and bases on the property of reverse k-nearest neighbor. In this section, we review some related works about reverse k-nearest neighbor and kNN join.

A. Reverse k-Nearest Neighbor

The reverse k-nearest neighbor (RkNN) problem has been first proposed in [22] and has been paid increasing attention to in the recent years [22], [28], [31], [27], [21], [23], [16].

Existing studies all focus on the single RkNN query. Various methods have been proposed for its efficient processing and can be divided into two categories: *pre-computation* methods and *space pruning* methods.

Pre-computation methods [22], [31] pre-compute and store the nearest neighbors of each point in a dataset in index structures, i.e., the RNN-tree [22] and the Rdnn-tree[31]. With the saved pre-computed nearest neighbor information, an RNN query is answered by a *point enclosure query* [22] that retrieves points p which have the query point fall within the sphere centered at p and of the radius dnn. A shortcoming of *pre-computation* methods is that they cannot answer an RkNN query unless the corresponding k-nearest neighbor information is available. The preprocessing of k-nearest neighbor information is actually a kNN join.

Space pruning methods [25], [29], [27] utilize the geometry properties of RNN to find a small number of data points as candidates and then verifies them with NN queries or range queries. Existing proposed algorithms are all based on the R-tree [4]. SAA [27] makes use of the *bounded output* property, e.g. for an RNN query in the 2-dimensional space, a query point q has at most 6 RNN [26]. SFT [25] is based on the assumption that RkNN and KNN are correlated, that is, an RkNN of q is expected to be one KNN of q, where K is a value bigger than k. The most recent work TPL [29] makes use of the *half-planes* pruning strategy, that is, if we divide the data space into two half-planes by the perpendicular bisector between q and an arbitrary data point p, any point in the half plane of p cannot be an RNN of q. *Space pruning* methods are flexible because they do not need to pre-compute the nearest neighbors information. However, they are very expensive when data dimensionality is high or the value k is big.

There are other RkNN related works include the bi-chromatic RNN query [28], the reverse nearest neighbor aggregates over data streams [23] and reverse nearest neighbor queries of moving objects [21]. Our work is the first that applying RkNN into data mining tasks.

B. kNN Join

k-nearest neighbor join (kNN-join) is a new operation proposed recently [6]. The operation combines each point of one dataset with its k-nearest neighbors in another dataset. It is identified that many standard algorithms in almost all stages of knowledge discovery process can be accelerated by including kNN-join as a primitive operation [30]. For examples, the each iteration of the well-known k-means clustering process, the first step of LOF [9] (a density-based outlier detection method), the kNN-graph (a graph linking each point of a dataset to its k-nearest neighbors) construction of the hierarchical clustering method chameleon [20].

Compared to the traditional point-at-a-time approach that computes the k-nearest neighbors for all data points one by one, the set oriented kNN-join can accelerate the computation dramatically[7].

The MuX kNN-join [6], [7] and the Gorder kNN-join are two up-to-date methods specifically designed for kNN-join of high-dimensional data. MuX [8] is essentially an R-tree based method designed to satisfy the conflicting optimization requirements of CPU and I/O cost. It employs large-sized pages (the hosting page) to optimize I/O time and uses the secondary structure, the buckets which are MBRs (minimum bounding boxes) of much smaller size, to partition the data with finer granularity so that CPU cost can be reduced. MuX iterates over the R pages, and for R page in the memory, potential kNN-joinable pages in S are retrieved through MuX index on S and searched for k-nearest neighbors. Since MuX makes use of an index to reduce the number of data pages retrieved, it suffers as an R-tree based algorithm and its performance degrades rapidly when the data dimensionality increases.

Gorder kNN-join [30] is a non-index approach. It optimizes the block nested loop join with efficient data scheduling and distance computation filtering by sorting data into the G-order. The dataset is then partitioned into blocks that are amenable for efficient scheduling for join processing and the *scheduled block nested loop join* is applied to find the k-nearest neighbors for each block of R data points. Gorder is efficient due to the following factors: (1) It inherits the strength of the block nested loop join in being able to reduce random reads. (2) It prunes away unpromising data blocks from probing to save both I/O and similarity computation costs by exploiting the property of the G-ordered data. (3) It utilizes a *two-tiers partitioning strategy* to optimize I/O and CPU time separately. (4) It reduces distance computational cost by pruning redundant computation based the distance of fewer dimensions. Study in [30] shows that Gorder is efficient and scalable with regard to both data dimensionality and size, and outperforms MuX by a significant margin. Hence, we utilize Gorder in the kNN join procedure of BORDER.

VI. Conclusion

In this paper, we introduce the problem of boundary point detection. The knowledge of boundary points can help with data mining tasks such as data preparation for clustering and classification. We propose a novel method BORDER (a BOundaRy points DEtectoR) which employs the state-of-the-art kNN join technique and makes use of the property of the RkNN. Experimental study demonstrates BORDER detects boundary points efficiently and effectively.

VII. Acknowledgments

This work was supported by AStar-University grant R-252-000-195-305.

References

[1] C. C. Aggarwal and P. S. Yu. Outlier detection for high dimensional data. In *Proc. of SIGMOD*, 2001.

[2] V. Barnett and T. Lewis. *Outliers in Statistical Data*. John Wiley and Sons, 1994.

[3] M. Basseville and I.V. Nikiforov. *Detection of abrupt changes*. P T R Prenstice Hall, 1993.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. of ACM SIGMOD*, pages 322–331. 1990.

[5] C. Böhm, S. Berchtold, and D.A. Keim. Searching in high dimensional spaces: index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[6] C. Böhm and F. Krebs. The k-nearest neighbor join: Turbo charging the kdd process. *Knowledge and Information Systems (KAIS)*.

[7] C. Bohm and F. Krebs. Supporting kdd applications by the k-nearest neighbor join. In *Proc. of DEXA*, pages 504–516, 2003.

[8] C. Böhm and H.-P. Kriegel. A cost model and index architecture for the similarity join. In *Proc. of ICDE*, pages 411–420, 2001.

[9] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: identifying density-based local outliers. In *Proc. of ACM SIGMOD*, pages 93–104. 2000.

[10] B.E. Brodsky and B.S. Darkhovsky. *Nonparametric methods in change-point problems*. Kluwer Academic Publishers, 1993.

[11] K. Chakrabarti and S. Mehrotra. Local dimensionality reduction: A new approach to indexing high dimensional spaces. In *Proc. of VLDB*, pages 89–100, 2000.

[12] Martin Ester, Hans-Peter Kriegel, Jorg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *SIGKDD*, pages 226–231, 1996.

[13] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. *Advances in Knowledge Discovery and Data Mining*. AAAI Press, 1996.

[14] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of ACM SIGMOD*, pages 47–57, 1984.

[15] J. Han and M. Kamber. *Data Mining Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.

[16] W. Hsu, M.-L. Lee, B. C. Ooi, P. K. Mohanty, K. L. Teo, and C. Xia. Advanced database technologies in a diabetic healthcare system. In *Proc. of VLDB*, pages 1059–1062, 2002.

[17] Data Mining Software in Java. *http://www.cs.waikato.ac.nz/ml/weka/*.

[18] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[19] H. Jin, B. C. Ooi, H. T. Shen, C. Yu, and A. Y. Zhou. An adaptive and efficient dimensionality reduction algorithm for high-dimensional indexing. In *Proc. of ICDE*, pages 87–98, 2003.

[20] G. Karypis, E.-H. Han, and V. Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75, 1999.

[21] G. Kollios, D. Gunopulos, and V. J. Tsotras. Nearest neighbor queries in a mobile environment. In *Spatio-Temporal Database Management*, pages 119–134, 1999.

[22] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *Proc. of ACM SIGMOD*, pages 201–212, 2000.

[23] F. Korn, S. Muthukrishnan, and D. Srivastava. Reverse nearest neighbor aggregates over data streams. In *Proc. of VLDB*, 2002.

[24] Lajos Horváth M. Csörgö. *Limit Theorems in Change-Point Analysis*. Wiley, 1997.

[25] A. Singh, H. Ferhatosmanoglu, and A. Ş. Tosun. High dimensional reverse nearest neighbor queries. In *Proc. of CIKM*, pages 91–98, 2003.

[26] M. Smid. Closest point problems in computational geometry. In *Handbook on Computational Geometry*. Elsevier Science Publishing, 1997.

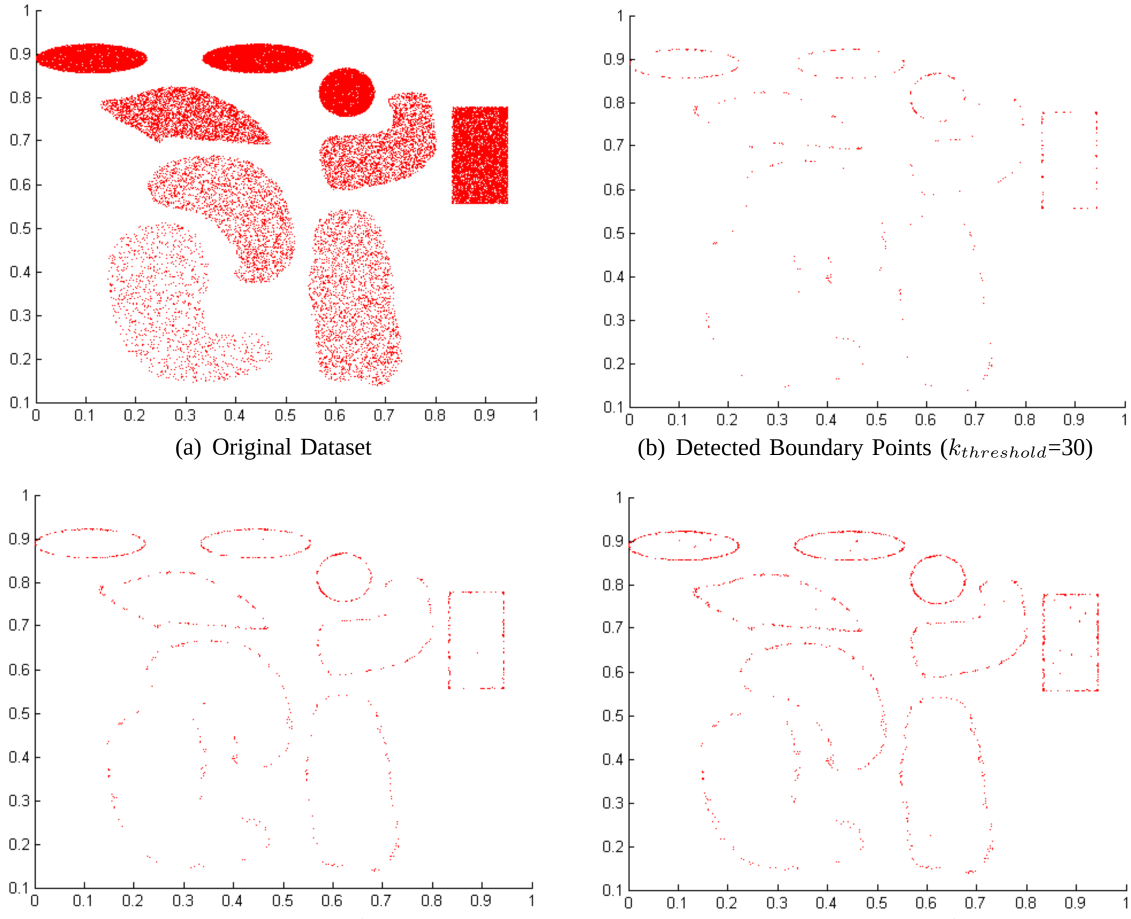
[27] I. Stanoi, D. Agrawal, and A. E. Abbadi. Reverse nearest neighbor queries for dynamic databases. In *Proc. of ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 44–53, 2000.

[28] I. Stanoi, M. Riedewald, D. Agrawal, and A. El Abbadi. Discovery of influence sets in frequently updated databases. In *Proc. of VLDB*, pages 99–108, 2001.

[29] Y. Tao, D. Papadias, and X. Lian. Reverse knn search in arbitrary dimensionality. In *Proc. of VLDB*, pages 744–755, 2004.

[30] C. Xia, H. Lu, B. C. Ooi, and J. Hu. Gorder: An efficient method for knn join processing. In *Proc. of VLDB*, 2004.

[31] C. Yang and K.-I. Lin. An index structure for efficient reverse nearest neighbor queries. In *Proc. of ICDE*, pages 485–492, 2001.



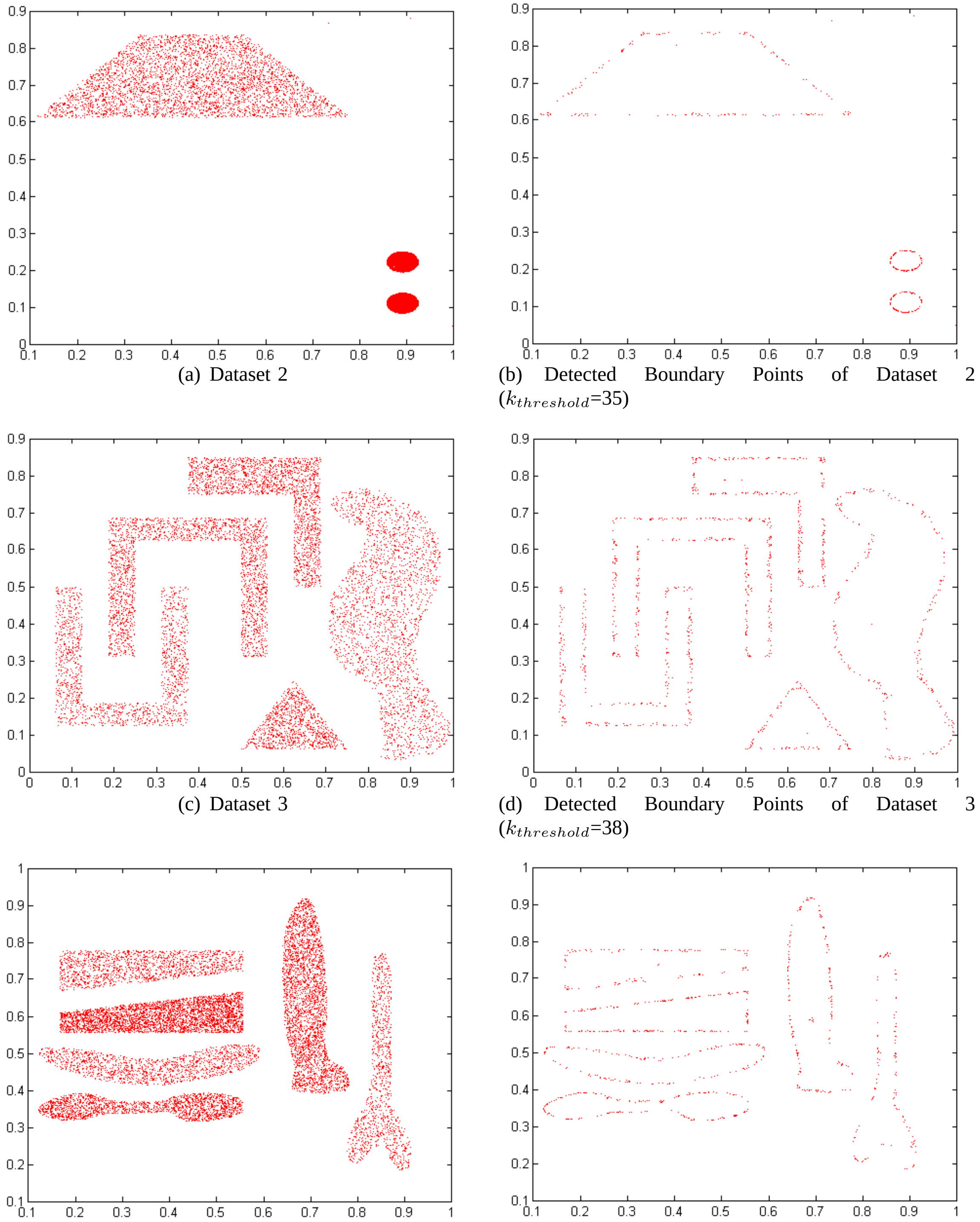
(a) Original Dataset

(b) Detected Boundary Points ($k_{threshold}$=30)

(c) Detected Boundary Points ($k_{threshold}$=33)

(d) Detected Boundary Points ($k_{threshold}$=35)

Fig. 10. Incremental output of detected boundary points of dataset 1.



(a) Dataset 2

(b) Detected Boundary Points of Dataset 2 ($k_{threshold}$=35)

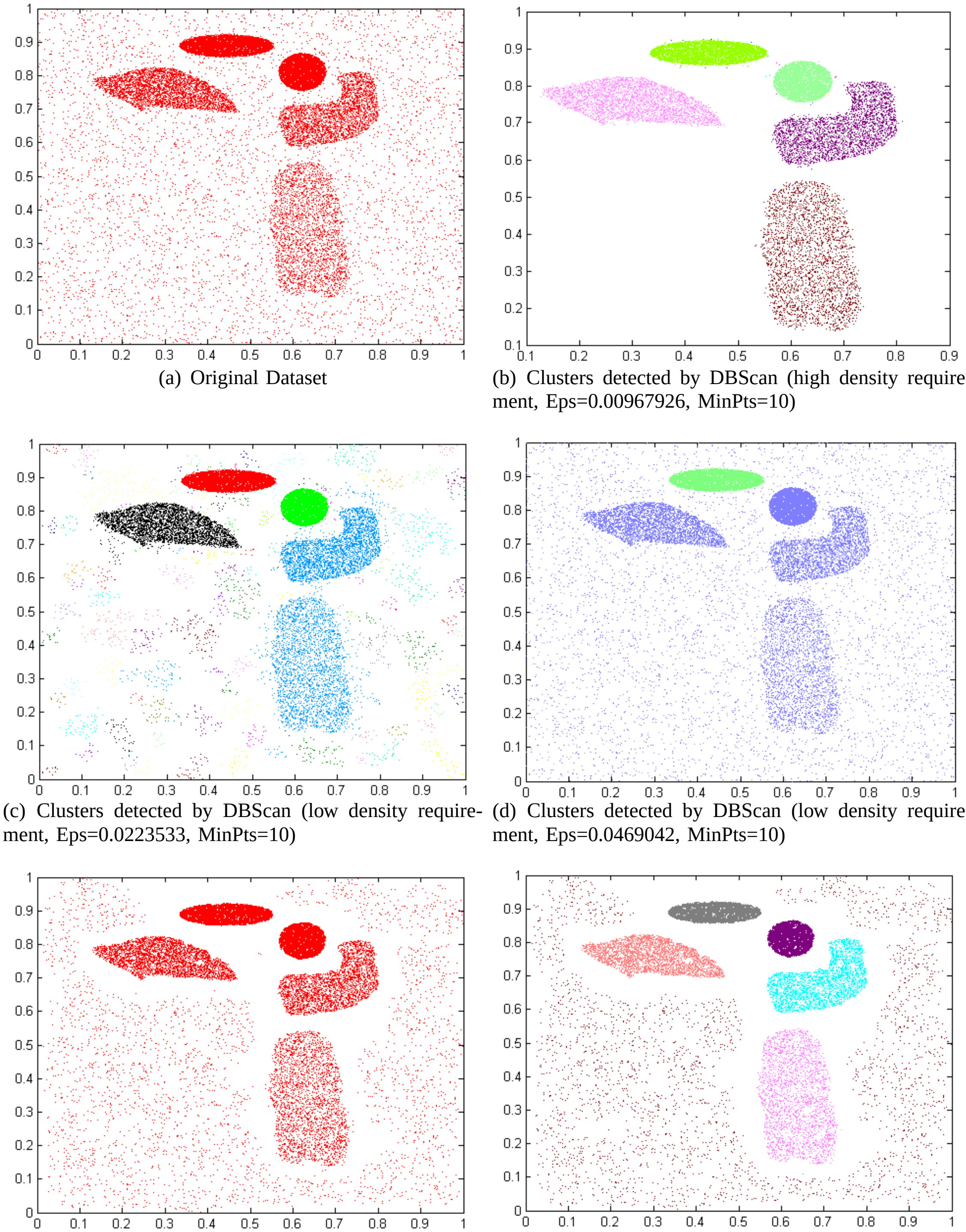
(c) Dataset 3

(d) Detected Boundary Points of Dataset 3 ($k_{threshold}$=38)

(e) Dataset 4

(f) Detected Boundary Points of Dataset 4 ($k_{threshold}$=35)

Fig. 11. Study on other datasets.



(a) Original Dataset

(b) Clusters detected by DBScan (high density requirement, Eps=0.00967926, MinPts=10)

(c) Clusters detected by DBScan (low density requirement, Eps=0.0223533, MinPts=10)

(d) Clusters detected by DBScan (low density requirement, Eps=0.0469042, MinPts=10)

(e) After boundary points are removed

(f) Clusters detected by DBScan after boundary points are removed(Eps=0.0469042, MinPts=10)

Fig. 12. Study on mixed clustered datasets.

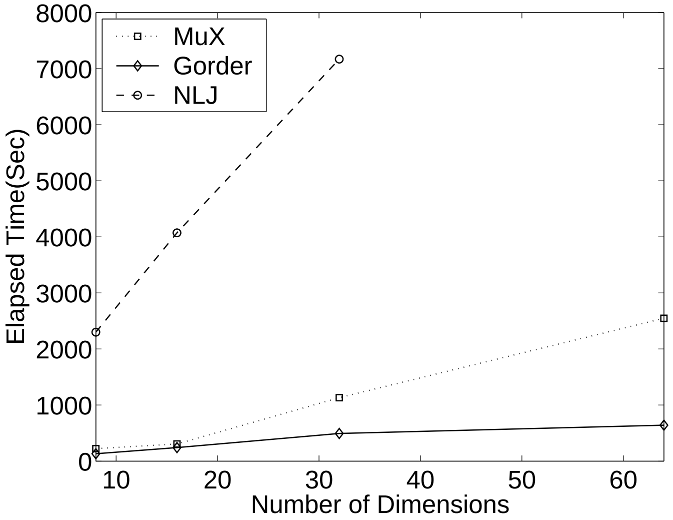


Fig. 13. Effect of dimensionality (100k clustered dataset)

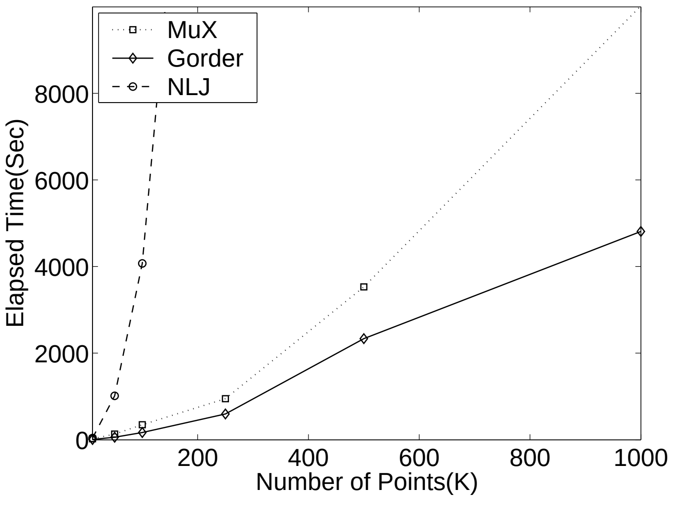


Fig. 14. Effect of data size (16-dimensional clustered datasets)