# A Novel Method for Detecting Outlying Subspaces in Highdim ensional Databases Using Genetic Algorithm

Ji Zhang, Qigang Gao
Faculty of Computer Science
Dalhousie University
Halifax, Nova Scotia, Canada
{jiz, qggao}@cs.dal.ca

Hai Wang
Sobey School of Business
Saint Mary's University
Halifax, Nova Scotia, Canada
hwang@smu.ca

## Abstract

*Detecting outlying subspaces is a relatively new research problem in outlier-ness analysis for high-dimensional data. An outlying subspace for a given data point $p$ is the subspace in which $p$ is an outlier. Outlying subspace detection can facilitate a better characterization process for the detected outliers. It can also enable outlier mining for high-dimensional data to be performed more accurately and efficiently. In this paper, we proposed a new method using genetic algorithm paradigm for searching outlying subspaces efficiently. We developed a technique for efficiently computing the lower and upper bounds of the distance between a given point and its $k^{th}$ nearest neighbor in each possible subspace. These bounds are used to speed up the fitness evaluation of the designed genetic algorithm for outlying subspace detection. We also proposed a random sampling technique to further reduce the computation of the genetic algorithm. The optimal number of sampling data is specified to ensure the accuracy of the result. We show that the proposed method is efficient and effective in handling outlying subspace detection problem by a set of experiments conducted on both synthetic and real-life datasets.*

## 1 Introduction

Outlier detection is an important research problem in data mining that aims to find a specific number of objects that are considerably dissimilar, exceptional and inconsistent with respect to the majority records in the input databases. Numerous research work in outlier detection has been proposed such as the distribution-based methods [3][6], the distance-based methods [9][10][12], the density-based methods [2][8][13] and the clustering-based methods [1][5][7][11][15][19], etc.

In this paper, we focus on the problem of *outlying subspace detection* for high dimensional data, which is a complementary problem of outlier detection. The major task of outlying subspace detection is to find the subspaces (subsets of features) in which each data point exhibits significant deviation from the rest of population. An outlying subspace for a data point is a subspace (a subset of features) in which this data can be considered as an outlier. The problem of outlying subspace can be formulated as follows: *given a data point, find the subspaces in which this point is considerably dissimilar, exceptional or inconsistent with respect to the remaining population in the database* [14].

Outlier mining can be benefited from outlying subspace detection in many aspects. First, outlying subspace detection can contribute to a better characterization of the outliers detected. The characterization of outliers mainly involves presenting the subspaces in which these outliers exist. In high-dimensional space, it is important to not only mine outliers but also find the *context* in which these outliers exist. In conventional outlier detection methods, each detected outlier can only be characterized by the subspace chosen by users that is used for outlier detection. Outlying subspace detection makes it possible to give a more informative characterization of the outliers detected by finding their outlying subspaces. Second, outlying subspace detection can enable outlier mining to be performed more accurately. It can allow outliers to be detected from more than one subspaces. This is important to many practical applications. For instance, in credit card fraud detection, online credit card transactions are needed to be scanned in order to determine whether each transaction is legal or fraudulent. The conventional outlier detection methods usually rely on a pre-specified feature subspace to detect outliers, which may cause them to miss many potential outliers hidden in other feature subspaces and fail to raise necessary alarms. Therefore, outlying subspace analysis plays a crucial role for outlier mining in this sitation to identify the correct feature subspaces in which outliers can be accurately mined. Finally, outlying subspace detection can help outlier detection methods to mine outliers in high-dimensional space more efficiently. It is an important intermediate step in *example-based* outlier mining, which detects outliers based on a set of outlier examples supplied by domain experts [16][17]. The basic idea of this method is to find the outlying subspaces of these outlier examples, from which more outliers that have similar outlier-ness characteristics to the given examples can be found more efficiently by only investigating the detected outlying subspaces.

Unfortunately, due to the exponential growth of the number of subspaces with respect to the dimension of the dataset, the problem of outlying subspace detection is NP-hard by nature. The straightforward exhaustive search is apparently infeasible to this problem, especially for high-dimensional datasets. In response to the inherent hardness of this problem, the state-of-the-art methods were proposed to employ heuristics in speeding up the search process in order to render this problem tractable. Zhang et al. proposed a dynamic outlying subspace search algorithm that utilizes a sample-based learning process to efficiently identify the outlying subspaces for the given points [14][18]. Two heuristic pruning strategies employing the upward and downward closure property are devised to reduce search space. Its major drawbacks, however, lie in the unsatisfactory accuracy of the metric used for measuring outlying degree of points in subspaces, the binary fashion of its result and the difficulty in specifying the distance threshold. Zhu et al. draw on a genetic algorithm to solve the example-based outlier detection problem [16][17]. The major limitation of this method is that it is computationally expensive to compute the outlying degree of points in subspaces. This is because that it uses a cell-based partitioning technique that scales poorly in high-dimensional space.

In this paper, we develop a method based on genetic algorithm to solve the outlying subspace detection problem that well copes with the drawbacks of the existing methods. The main contributions of this paper are summarized as follows:

1. A new metric, called *Subspace Outlying Factor* (SOF), is developed for measuring the outlying degree of each data point in different subspaces. Based on SOF, a new definition of outlying subspace, called *SOF Outlying Subspaces*, is proposed. The parameters used in defining SOF Outlying Subspaces are easy to be specified, and do not require any prior knowledge about the data distribution of the dataset;

2. A genetic algorithm-based method is proposed for outlying subspace detection. The upward and downward closure property is no longer required in our GA-based method, and the detected outlying subspaces can be ranked based on their fitness function values;

3. The concepts of the lower and upper bounds of $D^k$, the distance between a given point and its $k^{th}$ nearest neighbor, are proposed. These bounds are used for a significant performance boost in our method. We propose a technique to compute these bounds efficiently using the so-called $k$NN Look-up Table;

4. The random sampling technique is utilized in our method to further speed up the computation. The optimal number of sampling data is specified, and a novel genetic algorithm is developed to combine incremental data sampling and subspace fitness evaluation;

5. Last but not the least, we show that the proposed method is efficient and effective in handling outlying subspace detection problem through experiments conducted on both synthetic and real-life datasets.

## 2 Problem Formulation

To define outlying subspace, we need to first devise the metric for measuring outlier-ness of the given data point in different subspaces. In this work, we use $D^k$, the distance between a point and its $k^{th}$ nearest neighbor, in our outlier-ness metric, called *Subspace Outlying Factor (SOF)*. Mathematically, the SOF of a subspace $s$ w.r.t a given point $p$ is defined as the ratio of $D^k(p)$ in $s$ against the averaged $D^k$ in $s$ for points in the dataset $\mathcal{D}$, i.e.

$$SOF(s,p) = \frac{D^k_s(p)}{\overline{D^k_s(\mathcal{D})}} \tag{1}$$

Intuitively, the higher the ratio is, the higher the $D^k$ of $p$ is when compared to other points, therefore the higher outlier-ness of $p$ is and vice versa. Our definition of SOF leads to the following definition of *SOF Outlying Subspaces*:

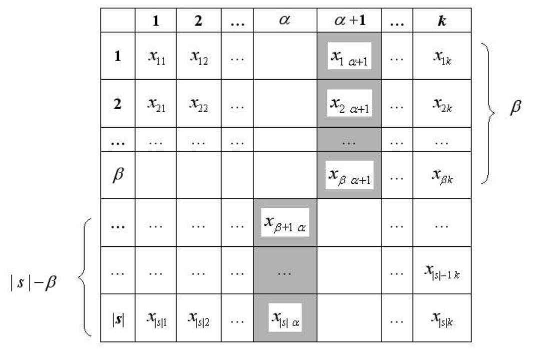
**Definition 1.** $SOF$ **Outlying Subspaces:** Given an input dataset $\mathcal{D}$, parameters $n$ and $k$, a subspace $s$ is a $SOF$ *Outlying Subspace* for a given data point $p$ if there are no more than $n-1$ other subspaces $s'$ such that $SOF(s',p) > SOF(s,p)$.

The above definition is equivalent to say that the top $n$ subspaces having the largest SOF values are considered to be outlying subspaces.

## 3 Lower and Upper Bounds of $D^k$

As the lower and upper bounds of $D^k$ of data points are established by means of their respective $k$NN Lookup Table, we therefore first introduce $k$NN Lookup Table prior to our discussion on the bounds of $D^k$.

**Definition 2: Full $k$NN Lookup Table:** A Full $k$NN Lookup Table of a data $p$, denoted as $\mathcal{T}^p$, is a $\varphi \times k$ table



| | 1 | 2 | … | $\alpha$ | $\alpha+1$ | … | $k$ |
|---|---|---|---|---|---|---|---|
| 1 | $x_{11}$ | $x_{12}$ | … | | $x_{1\ \alpha+1}$ | … | $x_{1k}$ |
| 2 | $x_{21}$ | $x_{22}$ | … | | $x_{2\ \alpha+1}$ | … | $x_{2k}$ |
| … | … | … | … | | … | … | … |
| $\beta$ | | | | | $x_{\beta\ \alpha+1}$ | … | $x_{\beta k}$ |
| … | … | … | … | $x_{\beta+1\ \alpha}$ | | … | … |
| … | … | … | … | … | | … | $x_{\vert s\vert-1\ k}$ |
| $\vert s\vert$ | $x_{\vert s\vert 1}$ | $x_{\vert s\vert 2}$ | … | $x_{\vert s\vert\ \alpha}$ | | … | $x_{\vert s\vert k}$ |

**Figure 1. Partial $k$NN Lookup Table of a data point**

containing information about its $k$ nearest neighbors in *each* single dimension of full data space with $\varphi$ dimensions. The entry $x_{ij}$ of the table represents the $j^{th}$ nearest neighbor of $p$ in the $i^{th}$ dimension, where $1 \le i \le \varphi$ and $1 \le j \le k$.

It is important to note that the lower and upper bounds of $D^k(p)$ will vary in different subspaces and only a portion of $\mathcal{T}^p$ is actually used to construct the bounds of $D^k(p)$ in a particular subspace $s$. That is, only the dimensions relevant to $s$ are needed to be considered. As such, we propose the notion of Partial $k$NN Lookup Table that is defined as a view of the corresponding full $k$NN Lookup Table.

**Definition 3: Partial $k$NN Lookup Table:** A Partial $k$NN Lookup Table of a data $p$ with respect to a subspace $s$, denoted as $\mathcal{T}_s^p$, is a $|s| \times k$ *view* (logical table) of the full $k$NN Lookup Table of $p$, with each entry $x_{ij}$ being the $j^{th}$ nearest neighbor of $p$ in the $i^{th}$ dimension, where $d_i \in s$, $1 \le i \le |s|$ and $1 \le j \le k$. $|s|$ is the number of dimensions of $s$. Apparently, the Partial $k$NN Lookup Table is a selection of its corresponding full $k$NN Lookup Table, i.e. $\mathcal{T}_s^p = \sigma_{di \in s}\mathcal{T}^p$.

## 3.1 Lower Bound of $D^k$

Given a subspace $s$ and the number of nearest neighbor $k$, we will first calculate the following two constructs, $\alpha$ and $\beta$. These two constructs will be utilized in constructing the lower bound of $D^k$ of $p$ in $s$. $\alpha$ and $\beta$ are defined as follows:

$$\begin{cases} \alpha = \left\lfloor \frac{k-1}{|s|} \right\rfloor + 1 \\ \beta = (k-1) \bmod |s| \end{cases} \tag{2}$$

where $\left\lfloor \frac{k-1}{|s|} \right\rfloor$ denotes the maximum integer that does not exceed $\frac{k-1}{|s|}$.

When $\alpha$ and $\beta$ are available, we sort $\mathcal{T}_s^p$ based on the values of $D_{d_i}^{\alpha}(p)$, the distance between $p$ and its $\alpha^{th}$ nearest neighbor in $d_i$ ($d_i \in s$) and select the $\beta$ dimensions that have the lowest $D_{d_i}^{\alpha}(p)$ values which are presented in set $d'$:

$$d' = \{d_1, d_2, \ldots, d_\beta\} \tag{3}$$

Then we define the *Lower Bound Set* ($LBS$) of $p$ in $s$ as

$$LBS_s(p) = \{e_1, e_2, \ldots, e_{|s|}\} \tag{4}$$

where each element $e_i$ ($1 \le i \le |s|$) is a positive integer and specified as follows:

$$e_i = \begin{cases} \alpha + 1, \text{ if } d_i \in d' \\ \alpha, \quad\;\;\; \text{ if } d_i \notin d' \end{cases} \tag{5}$$

The lower bound of $D^k(p)$ for a point $p$ in $s$, denoted as $LB_s(D^k(p))$, is given as

$$LB_s(D^k(p)) = \sqrt{\frac{\sum_{i=1}^{|s|} D_{d_i}^{e_i}(p)^2}{|s|-1}}, e_i \in LBS_s(p) \tag{6}$$

where $D_{d_i}^{e_i}(p)$ denotes the distance between $p$ and its $e_i^{th}$ nearest neighbor in dimension $d_i \in s$.

Figure 1 presents the Partial $k$NN Lookup Table of a point. All the elements in $LBS_s(p)$ has been highlighted using darker background in the table. As we discussed earlier, there are $|s|$ elements in $LBS_s(p)$; the first $|s| - \beta$ elements come from Column $\alpha$ and another $\beta$ elements come from Column $\alpha + 1$. These $|s|$ elements in $LBS_s(D^k(p))$ form a *lower bound frontier* in the Partial $k$NN Lookup Table.

**Lemma 1**: $\sqrt{\frac{\sum_{i=1}^{|s|} D_{d_i}^{e_i}(p)^2}{|s|-1}}, e_i \in LBS_s(p)$ is the lower bound of $D^k(p)$ for a point $p$ in subspace $s$.

**Proof:** Let $NNIndex_{(p,d_i)}(q)$ be nearest neighbor index number of point $q$ with respect to $p$ in dimension $d_i$ of subspace $s$. For instance, if $q$ is the 10th nearest neighbor of $p$ in the 5th dimension of $s$, then $NNIndex_{(p,d_5)}(q) = 10$.

The total number of unique points $q$ such that $NNIndex_{(p,d_i)}(q) < e_i$ for any dimension $d_i$ in $s$, $1 \le i \le |s|$ (i.e. falling to the left side of the lower bound frontier of the table) is no more than $k-1$. Without losing generality, let us suppose that there are $t$ such unique points, $(1 \le t \le k-1)$.

For any point $q$ satisfying $NNIndex_{(p,d_i)}(q) \ge e_i$ for all the dimensions $d_i$ of $s$, $1 \le i \le |s|$ (i.e. locating to the right of the lower bound frontier of the table), we have $dist_{d_i}(p,q) \ge D_{d_i}^{e_i}(p)$ for each $d_i \in s$. Therefore, $dist_s(p,q) \ge \sqrt{\frac{\sum_{i=1}^{|s|} D_{d_i}^{e_i}(p)^2}{|s|-1}}$.

Suppose, among the $t$ unique points locating to the left of the lower bound frontier of the table, there are $t'$ points $q$ satisfying $NNIndex_{(p,s)}(q) \le t$ $(0 \le t' \le t)$. Then, the $(t'+1)^{th}$ nearest neighbor of $p$ in $s$ should locate to the right of the lower bound frontier and $D_s^{t'+1}(p) \ge \sqrt{\frac{\sum_{i=1}^{|s|} D_{d_i}^{e_i}(p)^2}{|s|-1}}$. Since $1 \le t \le k-1$ and $t' \le t$, thus,

$t' + 1 \leq k$ and $D_s^{t'+1}(p) \leq D_s^k(p)$. Given $D_s^{t'+1}(p) \geq \sqrt{\frac{\sum_{i=1}^{|s|} D_{d_i}^{e_i}(p)^2}{|s|-1}}$ and $D_s^{t'+1}(p) \leq D_s^k(p)$, then we have $D_s^k(p) \geq \sqrt{\frac{\sum_{i=1}^{|s|} D_{d_i}^{e_i}(p)^2}{|s|-1}}$, as required. ■

## 3.2 Upper Bound of $D^k$

To compute the upper bound of $D^k(p)$ of $p$ in $s$, we need to first obtain the *Upper Bound Set* (UBS) for $p$ in $s$. It is defined as follows:

$$UBS_s(p) = \cup_{i=1}^{|s|} KNNSet_{d_i}(p), d_i \in s \tag{7}$$

where $KNNSet_{d_i}(p)$ denotes the set of $k$NNs of $p$ in dimension $d_i$ and it represents each row of $\mathcal{T}_s^p$. Obviously, we have $k \leq |UBS_s(p)| \leq k|s|$. $|UBS_s(p)| = k$ when the $k$NNs of $p$ in each dimension $d_i \in s$ are identical, whereas $|UBS_s(p)| = k|s|$ when there are no duplicates in $\mathcal{T}_s^p$. In computing the lower bound of $D^k(p)$, $UBS_s(p)$ is used to store the result of union operation on all $k$NN sets of $p$ in each single dimension.

Let point $q$ be the $k^{th}$ nearest neighbor of $p$ in $UBS_s(p)$. The upper bound of $D^k$ of $p$ in $s$ is defined as the distance between $p$ and $q$ in $s$ as

$$UB_s(D^k(p)) = dist_s(p, q) \tag{8}$$

**Lemma 2**. Let point $q$ be the $k^{th}$ nearest neighbor of $p$ in $UBS_s(p)$, then $dist_s(p, q)$ is the upper bound of $D^k$ of $p$ in $s$.

**Proof:** For two sets $set_1$ and $set_2$ such that $set_1 \subseteq set_2$ and $|set_2| \geq |set_1| \geq k$, if $q_1$ and $q_2$ are the $k^{th}$ nearest neighbor of $p$ in $set_1$ and $set_2$, respectively, then we have $dist_s(p, q_1) \geq dist_s(p, q_2)$. This is because that $D^k$ is monotonically decreasing as the set of points we examine gets larger.

Now let $set_1$ and $set_2$ be instantiated as $set_1 = UBS_s(p)$ and $set_2 = \mathcal{D}$, where $\mathcal{D}$ denotes the whole dataset, we thus have $UBS_s(p) \subseteq \mathcal{D}$ and $|\mathcal{D}| \geq |UBS_s(p)| \geq k$. Based on the above discussion, we will have $dist_s(p, q_1) \geq dist_s(p, q_2)$, where $q_1$ and $q_2$ are the $k^{th}$ nearest neighbor of $p$ in $UBS_s(p)$ and $\mathcal{D}$, respectively. Therefore, $dist_s(p, q_1)$ is the upper bound of $D^k$ of $p$ in $s$, as desired. ■

## 3.3 Approximation of Subspace Outlying Factor by Using Bounds of $D^k$

Let $\overline{LB_s(D^k, \mathcal{D})}$ and $\overline{UB_s(D^k, \mathcal{D})}$ be the average lower and upper bounds of $D^k$ in subspace $s$ for points in dataset $\mathcal{D}$. We define the minimum and maximum values for SOF of $p$ in $s$ as follows:

$$SOF_{min}(s, p) = \frac{LB_s(D^k(p))}{\overline{UB_s(D^k, \mathcal{D})}} \quad SOF_{max}(s, p) = \frac{UB_s(D^k(p))}{\overline{LB_s(D^k, \mathcal{D})}} \tag{9}$$

The approximated SOF of $s$ with respect to $p$ is computed by using the average of $SOF_{min}(s, p)$ and $SOF_{max}(s, p)$ as follows:

$$SOF_{app}(s, p) = \frac{SOF_{min}(s, p) + SOF_{max}(s, p)}{2} \tag{10}$$

## 3.4 Performance Improvement Using Approximation of SOF

As pointed out in [13], $k$NN search in subspace $s$ for all the $N$ points in the database requires a complexity of $O(k|s|N^2)$ when $s$ is of a high dimension.

If our approximation scheme of SOF is used, computing the lower bound of $D^k(p)$ in $s$ only requires sorting the $\alpha^{th}$ column of $\mathcal{T}_s^p$ and summing up $D_{d_i}^{e_i}(p)^2$ for each $d_i \in s$, with a complexity of $O(|s|log|s| + |s|)$. While for computing the upper bound of $D^k(p)$ in $s$, we need to find the $k^{th}$NN of $p$ in $UBS_s(p)$ with a maximum possible size of $k|s|$. Hence, the complexity will be $O(k|s| \cdot k|s|) = O(k^2|s|^2)$. In sum, the complexity of computing the bounds of $D^k$ for all the points in the database is $O((|s|log|s| + |s| + k^2|s|^2)N) = O(k^2|s|^2N)$.

By using our approximation technique, we are able to reduce the complexity in computing SOF of a subspace to a linear order with respect to $N$, leading to a computation saving by up to a factor of $\frac{N}{k|s|}$ compared to the case when no approximation is performed. Since $N >> |s|$ and $k$ is usually small in most cases, our approximation technique is thus able to achieve a significant performance improvement.

## 4 Construction of $k$NN Lookup Table

The $k$NN Look-up Table for a data point should be first constructed before its lower and upper bounds of $D^k$ can thereby be established. The key task involved in constructing the $k$NN Look-up Table of a data point is to find its $k$NNs in each single dimension of the full data space. To facilitate construction of the tables, we transform the original dataset into a few sorting lists, where the number of the sorting lists is equal to the number of dimensions of the dataset. Each sorting list is constructed based on the sorting order of data in each dimension. The lengths of all sorting lists are identical and equal to the number of data points in the original dataset. Each element in the sorting list has two fields: the ID and the value of a particular point in the related dimension. There can be a few ways to implement $k$NN search in the sorting lists. In this work, we will study three simple yet efficient methods, namely the list-based, block-based and tree-based methods.

**List-based Method** . $k$NN search can be performed directly on the sorting lists. We need to first locate $p$ in the sorting list and then perform $k$NN search for $p$. The preceding and subsequent $k$ points of $p$ are the candidates of its $k$NNs, from which $k$NNs of $p$ can be found.

Locating a point $p$ in any one sorting list needs a complexity of $O(log_2N)$ and finding $k$NNs in the sorting list requires another $O(k)$ computation. In sum, the computational complexity of employing the sorting list will be $O(log_2N + k)$. The space complexity of list-based method is $O(2N)$.

**Lemma 3:** If a sorting list is partitioned into blocks with an equal size $b$, then at most 3 blocks in the sorting list are needed to be searched for finding $k$NNs for any point $p$ if $b \geq k$. These 3 blocks are the block $B$ to which $p$ belong and the the two adjacent blocks of $B$.

**Proof:** Without losing generality, let us suppose that the sorting list is related to dimension $d_i$ ($1 \leq i \leq \varphi$) and the block index number of $B$ to which $p$ belongs is $i$. For any point $q$ in Block $(i-2)$, there are at least $k$ points in Block $i-1$ whose distance to $p$ in dimension $d_i$ is less than or equal to $dist_{d_i}(p,q)$. Likewise, for any point $q$ in Block $(i+2)$, there are at least $k$ points in Block $(i+1)$ whose distance to $p$ in dimension $d_i$ is less than or equal to $dist_{d_i}(p,q)$. Therefore, all $k$NNs of $p$ definitely fall into Block $(i-1)$, $i$ or $(i+1)$. ■

**Blockbased Method** . $k$NN search can also be performed on a block-by-block basis. The basic idea is to partition a sorting list into a number of blocks. These blocks are usually of an equal size and contain more than $k$ points. Each time, only a single block is evaluated. Let $B_{min}$ and $B_{max}$ be the minimum and maximum point values in the current block being loaded, respectively, and $p.value$ be the value of $p$ in the sorting list. If $B_{min} \leq p.value \leq B_{max}$, then this block is the one to which $p$ belongs. We can further locate $p$ in this block and perform $k$NN search for $p$. If $B_{min} \leq p.value \leq B_{max}$ is not met for the current block, then another block will be loaded for evaluation.

The computational complexity for finding the block to which $p$ belongs is $O(1)$ for the best case where $p$ is located in the first block we evaluate and $O(\frac{N}{b})$ for the worst case where all the blocks in the sorting list are exhausted in the search. The average-case complexity is thus $O(\frac{N+b}{2b})$. Also, the time complexity for locating $p$ in the block is $O(log_2b)$. Finding $k$NNs in the sorting list requires another $O(k)$ computation. The total time complexity is $O(\frac{N+b}{2b} + log_2b + k)$. Since at most 3 blocks in the sorting list need to be searched, the space complexity is therefore $O(6b)$.

**Treebased Method.** The third alternative is to utilize a tree structure to further index the data blocks in the sorting list. For the sake of simplicity, we construct *Binary Trees* for performance enhancement in $k$NN search. Binary tree is simple in structure yet very efficient in $k$NN search.

The binary tree we use for each sorting list is a balanced rooted tree $BT =< V, E >$, where $V$ is the node set and $E$ is the edge set. The nodes in $V$ are classified as the *block nodes* and the *indexing nodes*. The *block nodes* are the leaves of the binary tree that represent data blocks in a sorting list that each contains $b$ points, where $b \geq k$. Each data block in the leaf level will be represented by its minimum point value (the first value in the block if the sorting list is ordered in an ascending order) in the binary tree. The *indexing nodes* are other nodes in the tree primarily used for indexing the data blocks at the bottom level. The immediate indexing node of a block node takes the smallest value and the starting address of the data block. Other indexing nodes (excluding the root) will take the minimum value of its children, together with the addresses of its left and right children. The root will only stores the addresses of its two children.

$k$NN search for a point $p$ in a binary tree also takes two major steps. First, the binary tree is traversed top-down from the root until the block to which $p$ belongs is found. The moment an intermediate indexing node $a$ is reached, the following rule is used to decide the sub-tree for further traversal: *If $p.value \geq a.rightchild$, then right child of $a$ is chosen for traversal. Otherwise, the left child is selected for traversal.* When a bottom indexing node (i.e. the immediate parent of a block node) is reached, the block node to which it is pointing is referred and the whole data block is fetched. After the block to which $p$ belongs to has been found, the location of $p$ within the block will be determined and $k$NN search can be performed.

It will require $O(log_2\frac{N}{b})$ to traverse the binary tree downward from the root to find the block to which $p$ belongs. The complexity of locating $p$ in the block is $O(log_2b)$ and searching for $k$NNs of $p$ in 3 blocks requires a complexity of $O(k)$. In sum, the time complexity is $O(log_2\frac{N}{b} + log_2b + k) = O(log_2N + k)$.

The total number of *indexing nodes* in the tree is approximately $\frac{2N}{b}$. We load all the indexing nodes of the binary tree as they will be frequently used for $k$NN search. Since it is not required to load any data blocks until the block to which $p$ belong has been found and at most 3 blocks are needed to be loaded for $k$NN search for $p$, thus the space complexity of tree-based method is $O(\frac{2N}{b} + 6b)$.

## 5 Genetic Algorithm for Detecting Outlying Subspaces

In this section, we will elaborate on the design of the genetic algorithm for outlying subspace detection.

**Representation.** Our GA technique uses the standard binary individual encoding; all individuals are represented by strings with fixed and equal length $\varphi$, where $\varphi$ is the number of dimensions of the dataset. Using binary alphabet $\Sigma = \{0, 1\}$ for gene alleles, each bit in the individual will take on the value of "0" and "1", indicating whether or not its corresponding condition is selected, respectively.

```
Algorithm: OS_Detection($p$, $P$, $p_c$, $p_m$, $k$, $\epsilon$, $n_c$)
Input: a given point $p$, population size $P$, probability of crossover
$p_c$, probability of mutation $p_m$, number of nearest neighbors $k$,
convergence factor $\epsilon$ and number of candidate subspaces $n_c$.
Output: SOF Outlying Subspaces.
1.  $CompleteSet \leftarrow \emptyset$; $CandidateSet \leftarrow \emptyset$
2.  $S_{pop} \leftarrow$ initial population of $P$ subspaces;
3.  WHILE (evolution_stop_criterion=false) DO {
4.    FOR each individual $s$ in $S_{pop}$ DO
5.      CompFitness($s, p, k, \epsilon$);
6.      $CompleteSet \leftarrow CompleteSet \cup$ individuals and their
        respective SOF in $S_{pop}$
7.    $S_{pop} \leftarrow$ Selection($S_{pop}$);
8.    $S_{pop} \leftarrow$ Crossover($S_{pop}, p_c$);
9.    $S_{pop} \leftarrow$ Mutation($S_{pop}, p_m$); }
10. $CandidateSet \leftarrow$ top $n_c$ subspaces in $CompleteSet$;
11. $CandidateSet \leftarrow$ SubspaceRefine($CandidateSet$);
12. Return top $n$ individuals in $CandidateSet$;
```

**Figure 2. Genetic algorithm for outlying subspace detection**

**Fitness Function.** The fitness function used in the genetic algorithm is defined as the approximated SOF of subspace $s$ with respect to the given point $p$, as presented in Eq. (10), i.e.

$$f_{fit}(s) = SOF_{app}(s,p) = \frac{SOF_{min}(s,p) + SOF_{max}(s,p)}{2} \tag{11}$$

A higher value of $f_{fit}(s)$ indicates a fitter solution and vice versa. The definition of $f_{fit}(s)$ encourages the genetic algorithm to produce an increasing number of subspaces having high SOF values as evolution proceeds.

**Selection Operator.** In our work, *fitness-proportionate selection*, also known as roulette-wheel selection, is used to select fitter solutions in each step of the evolution. Fitness-proportionate selection is a stochastic selection method where the selection probability of a subspace is proportional to the value of its fitness function $f_{fit}(s)$.

**Search Operators.** The crossover and mutation used in this work is *single-point crossover* and *bit-wise mutation*. In our work, all the new individuals generated by crossover and mutation are of the same length, i.e. $\varphi$, as their parent(s), where $\varphi$ is number of dimensions of the input database. There are two associated probabilities, $p_c$ and $p_m$, used to determine the frequencies for applying crossover and mutation, respectively. Normally, we have $p_c >> p_m$, meaning that crossover is performed in a much higher frequency than mutation.

**Algorithm.** The framework of genetic algorithm for detecting outlying subspace is presented in Figure 2. $CompleteSet$ is the set used to maintain all the subspaces, together with their respective SOF, that have been evaluated in the genetic algorithm and $CandidateSet$ is the set used to only store the candidates of SOF Outlying Subspaces. The stopping criterion in the WHILE loop is usually that the number of generations performed has reached a pre-specified constant. $CandidateSet$ stores the top $n_c$ subspaces in $CompleteSet$ (line 10). In order to achieving good accuracy of the detected outlying subspaces, $n_c$ should be substantially larger than $n$. In Line 11, we perform subspace refinement (will be discussed in the sequel) and the top $n$ subspaces from all the candidates in $CandidateSet$ are returned as SOF Outlying Subspaces.

**The Subspace Refinement.** Since we approximate SOF in the genetic algorithm, the accuracy of computation is thus somehow limited. To address this problem, we can perform a refinement step on the candidate outlying subspaces in Line 11 of the genetic algorithm (Figure 2). Instead of using the lower and upper bounds of $D^k$ for a fast fitness approximation, the refinement step will compute the accurate SOF for all subspace candidates in $CandidateSet$ and the top $n$ outlying subspaces among them will be returned. A pruning optimization strategy can be devised based on the maximum value of SOF (i.e. $SOF_{max}$) of different subspaces to speed up the computation. The basic idea of this pruning optimization strategy is that, after $n$ subspace candidates have been evaluated, we start to maintain the minimum value of SOF for the top $n$ subspaces we have found thus far, denoted as $MinSOF_n$. Those subspaces satisfying $SOF_{max}(s,p) < MinSOF_n$ cannot become the top $n$ subspaces and can therefore be safely pruned. This is because that the value of $MinSOF_n$ is monotonically increasing as more subspaces are examined in the refinement step.

## 6 Random Sampling

The most computationally expensive step in our genetic algorithm lies in the fitness evaluation of individuals. This is because that the fitness evaluation for each subspace, either in approximated or accurate manner, involves scanning all the points in the dataset. This will be slow as the number of points in the dataset is usually large. To speed up fitness evaluation, we draw on random sampling technique so as to evaluate fitness of individuals only based on the random samples, rather than on the entire dataset.

By using sampling data, the average lower and upper bounds of $D^k$ in subspace $s$, used in SOF approximation, can be computed as follows:

$$\overline{LB_s(D^k, \mathcal{S})} = \frac{1}{N_{\mathcal{S}}} \sum_{i=1}^{N_{\mathcal{S}}} LB_s(D^k(sp_i))$$

$$\overline{UB_s(D^k, \mathcal{S})} = \frac{1}{N_{\mathcal{S}}} \sum_{i=1}^{N_{\mathcal{S}}} UB_s(D^k(sp_i)) \tag{12}$$

where $N_{\mathcal{S}}$ denotes the number of points in the sample $\mathcal{S}$ and $sp_i$ denotes the $i^{th}$ sampling point in $\mathcal{S}$, $1 \le i \le N_{\mathcal{S}}$.

Sampling can help improve the efficiency of our method significantly, but the quality of the result may be affected. In what follows, we will discuss convergence of the averaged lower and upper bounds of $D^k$ when the number of sampling data is increased.

Let $X$ be a variable that can represent the averaged lower or upper bound of $D^k$ for a set of data points in a subspace. Let us suppose that there are already $i-1$ sampling points and the $i^{th}$ sampling point is generated. From the convergence perspective, we want to ensure that $\exists \mu \ge 2$, where $\mu$ is a positive integer, for $\forall i \ge \mu$, we have

$$\frac{|\overline{X_i} - \overline{X_{i-1}}|}{\overline{X_{i-1}}} < \epsilon \tag{13}$$

where $\overline{X_i}$ denotes the average value of $X$ for all the first $i$ sampling points. $\epsilon$ is called convergence factor and is usually a small positive number (say 0.01). The ratio of $\frac{|\overline{X_i} - \overline{X_{i-1}}|}{\overline{X_{i-1}}} < \epsilon$ measures the degree to which the averaged value of $X$ changes due to the inclusion of the new (i.e. the $i^{th}$) sample point. Eq. (13) intuitively means that, when each newly generated sampling point does not considerably change the average value of $X$ after the sample reaches a certain size $\mu$, then we can claim that a convergence of $X$ value of data points have been achieved. $\overline{X_i}$ is defined based on $\overline{X_{i-1}}$ recursively as follows:

$$\begin{cases} \overline{X_1} = X_1; \\ \overline{X_i} = \frac{(i-1)\cdot \overline{X_{i-1}} + X_i}{i}, i \ge 2 \end{cases} \tag{14}$$

where $X_i$ denotes the $X$ value of the $i^{th}$ sample point.

Plug Eq. (14) into Eq. (13), we have

$$\frac{|\frac{(i-1)\cdot \overline{X_{i-1}} + X_i}{i} - \overline{X_{i-1}}|}{\overline{X_{i-1}}} < \epsilon$$

After simplification, we can get

$$\frac{|\frac{X_i}{\overline{X_{i-1}}} - 1|}{i} < \epsilon \tag{15}$$

**Lemma 4:** The minimum number for the sampling data is $\left\lceil \frac{(\frac{X_{max}}{X_{min}} - 1)}{\epsilon} \right\rceil$, where $\left\lceil \frac{(\frac{X_{max}}{X_{min}} - 1)}{\epsilon} \right\rceil$ denotes the minimum integer that is no less than $\frac{(\frac{X_{max}}{X_{min}} - 1)}{\epsilon}$, $X_{min}$ and $X_{max}$ denote the minimum and maximum values of $X$ for all the points in the dataset, respectively.

**Proof:** Based on Eq. (15), we need to have

$$i > \frac{|\frac{X_i}{\overline{X_{i-1}}} - 1|}{\epsilon} \tag{16}$$

to ensure the convergence of $\frac{|\overline{X_i} - \overline{X_{i-1}}|}{\overline{X_{i-1}}}$. Since we have $|\frac{X_i}{\overline{X_{i-1}}} - 1| \le \frac{X_{max}}{X_{min}} - 1$, therefore if we have $i > \frac{\frac{X_{max}}{X_{min}} - 1}{\epsilon}$ then Eq. (15) can always be satisfied. Therefore, the minimum number of sampling points required for $X$, denoted as $N^*_{sample}(X)$, is computed as

$$N^*_{sample}(X) = \left\lceil \frac{(\frac{X_{max}}{X_{min}} - 1)}{\epsilon} \right\rceil \tag{17}$$

As desired. ■

In order to produce sufficient sampling data to achieve convergence for both the lower and upper bounds of $D^k$, the optimal (minimum) number of sampling data in subspace $s$ is specified based on Eq (17) as follows:

$$N^*_{sample}(s) = max(N^*_{sample}(LB), N^*_{sample}(UB))$$

$$= \left\lceil \frac{max\left(\frac{LB_{max}(D^k)}{LB_{min}(D^k)}, \frac{UB_{max}(D^k)}{UB_{min}(D^k)}\right) - 1}{\epsilon} \right\rceil \tag{18}$$

where $LB_{max}(D^k)$ and $LB_{min}(D^k)$ are the maximum and minimum values of the lower bound of $D^k$ for all the points in the dataset, $UB_{max}(D^k)$ and $UB_{min}(D^k)$ are the maximum and minimum values of the upper bound of $D^k$ for all the points in the dataset.

Although the optimal number of sampling points has been explicitly specified in Eq. (18), we would face the following dilemma in practice when specifying its value: On one hand, the objective of performing data sampling is to achieve performance boost by only working on *a small portion* of the original dataset. On the other hand, however, the optimal number of sampling points cannot be specified without evaluating the *whole* dataset in order to find the global minima and maxima.

Due to the above dilemma, we propose a novel approach to progressively approximate the optimal sampling points in parallel with subspace fitness computation. The basic idea of this progressive approximation approach is to start with a set of sampling point with a minimum size (can be as small as 2 sampling points) and incrementally grow this set when necessary during the course of subspace evaluation in the genetic algorithm. Specifically, the approximation is performed progressively in the following two iterative steps:

1. The local minimum and maximum values of the lower and upper bounds of $D^k$ for the current sampling points are found and are used to compute the optimal number of sampling points $N^*_{sample}$;

2. If the number of current sampling points $N_{sample}$ is less than $N^*_{sample}$, then $N^*_{sample} - N_{sample}$ new sampling points will be generated.

The above two steps are repeated until $N^*_{sample} > N_{sample}$ is no longer met.

Having the sampling data for the first subspace, the sampling data to be used for subsequent subspaces will be generated in an *incremental* way. The sampling data for one subspace may or may not be large enough for achieving a convergence for the bounds of $D^k$ in the sampling data for another new subspace. To decide this, we need to compute $N^*_{sample}$ in the new subspace first and then test whether or not $N^*_{sample} \leq N_{sample}$ is met. If $N^*_{sample} \leq N_{sample}$ is met, indicating the current sampling data is sufficient to reach a convergence of the bounds for this new subspace, then we will just utilize the current set of sampling data for this new subspace without introducing any new ones. It is also possible that the current sampling data are not enough, thus more new sampling points will be generated for the new subspace until the convergence can be observed.

## 7 Experimental Results

We use both synthetic and real-life datasets for performance evaluation in our experiments. In the synthetic datasets, we are able to specify the number of instances (tuples) ($N$) and dimensions ($\varphi$) of the datasets generated. We also use four real-life multi- and high-dimensional datasets from the UCI machine learning repository in our experiments. These four datasets called *Letter Image* (D1, 16-dimensional), *Image Segmentation* (D2, 19-dimensional), *Ionosphere* (D3, 34-dimensional) and *Musk* (D4, 168-dimensional), respectively. No missing values will occur in all the synthetic and real-life datasets. As the experimental setup, we set the number of $SOF$ Outlying Subspaces returned in the end $n = 20$, the number of generations for the GA $N_g = 50$, the population size in each generation $P = 50$, the frequency of applying crossover $p_c = 0.8$ and the frequency of applying mutation $p_m = 0.2$.

### 7.1 Experimental Results on Synthetic Datasets

Experiments conducted on synthetic datasets are to test the effects of number of points $N$ and number of dimensions $\varphi$ of the dataset on the performance of our method.

**Effect of $N$ on constructing Full kNN Lookup Table.** $N$ determines the length of the sorting lists obtained from the original dataset, which will further affect the efficiency of $k$NN search of each point in constructing its $k$NN Lookup Table. In this experiment, the block size $b$ is set to be 1000 and 5000 for the block-based method and 1000 for the tree-based method (Note that the time complexity of tree-based method is independent of the block size $b$). The time is averaged over 20 given points for all the methods. Figure 3 presents the results. It shows that the list-based and tree-based methods are very close to each other in terms of running time and are more efficient than the block-based method. This is because that the time complexities of list-based and tree-based methods are both logarithmic w.r.t $N$ while that of the block-based method is approximately linear w.r.t $N$. Also, when block size $b$ is increased (say from 1000 to 5000 in this experiment), the complexity of the block-based method is decreased. In the extreme, when $b$ approaches $N$, the time complexity of the block-based method will become $O(log_2 N + k)$, which will be equivalent to the complexities of the list-based and tree-based methods.

**Effect of $N$ on SOF computation.** $N$ affects the efficiency of fitness evaluation for each subspace in the GA. When no approximation of SOF is used, the time complexity of fitness computation using the nested-loop method for $k$NN search is quadratic with respect to $N$. Nevertheless, the complexity can be reduced to a linear order of $N$ if our approximation scheme of SOF is employed. Moreover, if we choose to work on the sampling data for performance boost, then the execution time is independent of $N$ in any way. This is because that the number of sampling points we use in fitness evaluation for subspaces is only depended on the characteristics of data, as revealed in Eq. (18). This empirical analysis is confirmed in Figure 4. In this experiment, the execution time is the average time spent in evaluating each subspace. The results of this experiment illustrate that SOF approximation and sampling are very promising in boosting performance of our method. Under different $N$ values, our algorithm can run 2-20 times faster than the nested-loop method when using SOF approximation and can run 10-130 times faster when using both SOF approximation and random sampling.

**Effect of $\varphi$ on the search workload of the GA.** $\varphi$ determines the size of the search workload for subspaces, which is in an exponential order of $\varphi$. This does not necessarily mean that the running time of our algorithm will be exponential with respect to $\varphi$ whatsoever. The actual running time is depended on how the search workload in the GA is specified. More precisely, if we use a fixed number of generations and population size in each generation of the GA, i.e. *a fixed workload*, then the total search workload in the GA will be the same under different $\varphi$. Yet, using a fixed search workload in the GA for datasets with different dimensions may not an effective strategy. A *varied workload* scheme, in contrast, performs a search workload that is specified by a *workload function*. The workload function specified by the users is a monotonically increasing function of $\varphi$, reflecting the effect of $\varphi$ on the search workload of algorithm. The time complexity may now become exponential with respect to $\varphi$ as long as the workload function is an exponential function w.r.t $\varphi$. In our experiment, the search workload under the fixed workload scheme is set to be 2500 (50 generations with 50 individuals in each generation) and is stipulated by workload function $w = \varphi^2$ in

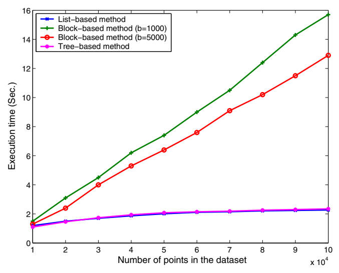


**Figure 3. Effect of $N$ on $k$NN Lookup Table Construction**

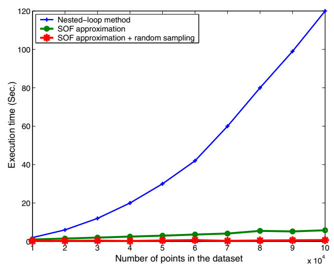


**Figure 4. Effect of $N$ on SOF Computation**

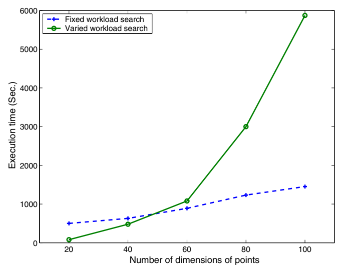


**Figure 5. Effect of $\varphi$ on our method**

the varied workload scheme. The running time of these two search schemes for detecting SOF Outlying Subspaces of 20 given data points are presented in Figure 5. The running time of our algorithm under fixed workload scheme scales linearly w.r.t $\varphi$ while that under varied workload scheme is in a quadratic order of $\varphi$.

## 7.2 Experimental Results on Real-life Datasets

Using the real-life multi- and high-dimensional datasets in UCI machine learning repository, we investigate the performance of our method in fitness convergence and subspace refinement.

**Fitness convergence study.** GA tends to produce an increasing number of fitter individuals as evolution proceeds, referring to as the phenomenon of convergence. In this experiment, we investigate the fitness convergence of our technique. For each generation, the number of individuals with relatively high fitness ($\geq 2.0$) are counted. As we can see from Figure 6 that the number of individuals with high fitness for the four datasets is increased as the GA evolves, which indicates a good convergence of our method. A good convergence is beneficial as this will enable our method to find good outlying subspaces w.r.t the given point without exploring a huge number of subspaces.

**Fitness boost by performing refinement in the GA**. We study the contribution of the subspace refinement step used in GA to enhancing fitness of the outlying subspaces detected, compared to the case when no refinement is involved. When no refinement is performed in the GA, the SOF Outlying Subspaces are the top $n$ subspaces in the $CandidateSet$ having the highest approximated SOF values. However, there top $n$ outlying subspaces chosen based on approximated SOF values may not be the true outlying subspaces having the high SOF values. The fitness improvement by using the refinement step is due to the extra computations performed on the subspaces in $CandidateSet$ in order to get their accurate SOF values and the top $n$ subspaces with the highest accurate SOF values are returned as SOF Outlying Subspaces. Figure 7 presents the results. The results demonstrate that the fitness gain by performing the refinement step ranges from 13% and 21% for the four datasets when compared with the case when no refinement is performed.

**Subspace pruning in refinement step of the GA.** In this experiment, we would like to study the advantage of employing the subspace pruning strategy, devised based on the bounds of $D^k$, in computation saving in the refinement step of the GA. The bounds of $D^k$ of subspaces help speed up our method by pruning away those subspaces in $CandidateSet$ that are definitely not relevant to the final SOF Outlying Subspace in the subspace refinement step. In this experiment, we set the number of subspaces in $CandidateSet$ as 1000 and compare it with the number of subspaces whose SOFs have actually been computed in the refinement step. From Figure 8, we can see that our pruning strategy is effective in greatly reducing the number of subspaces to be evaluated in the refinement step and such saving ranges from 19% to 41% for the four datasets.

## 8 Conclusions

In this paper, we address the problem of outlying subspace detection. Due to the inherent hardness of this problem, we utilize genetic algorithm as an efficient search method in this work. We proposed a new definition of outlying subspaces called *SOF Outlying Subspace*. The lower and upper bounds of $D^k$ for any a data point are revealed and three efficient methods are presented for computing the bounds by using the $k$NN Lookup Tables of data points. We also employ random sampling to significantly improve the performance of our method. The optimal number of sampling data ensuring a good approximation of SOF is given, and a novel genetic algorithm is developed for combining subspace fitness evaluation and data sampling. We present

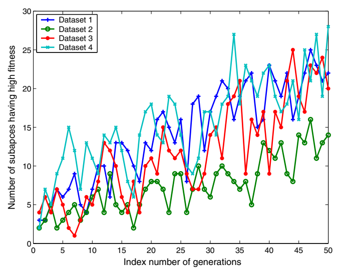


**Figure 6. Fitness convergence study**

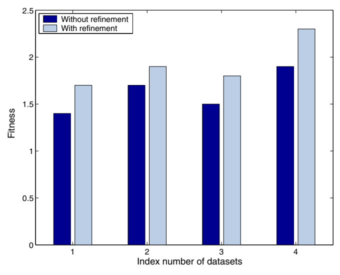


**Figure 7. Fitness improvement by subspace refinement**

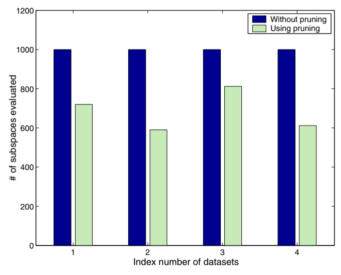


**Figure 8. .of subspaces evaluated in subspace refinement**

the experimental results of our method on both synthetic and real-life datasets. The results demonstrate the efficiency and effectiveness of our method in handling outlying subspace detection.

## Acknowledgment

This research work is supported in part by research grant of Natural Sciences and Engineering Research Council of Canada (Grant No.: 312423).

## References

[1] R. Agrawal, J. Gehrke, D. Gunopulos and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. *SIGMOD'98*, pp 94-105, 1998.

[2] M. Breuning, H-P. Kriegel, R. Ng, and J. Sander. LOF: Identifying Density-Based Local Outliers. *SIGMOD'00*, Dallas, Texas, pp 93-104, 2000.

[3] V. Barnett and T. Lewis. *Outliers in Statistical Data*. John Wiley, 3rd edition, 1994.

[4] L. Boudjeloud and F. Poulet. Visual Interactive Evolutionary Algorithm for High Dimensional Data Clustering and Outlier Detection. *PAKDD'05*, Hanoi, Vietnam, pp426-431, 2005.

[5] M. Ester, H-P. Kriegel, J. Sander, and X. Xu. A Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. *SIGKDD'96*, Portland, Oregon, USA, pp 226-231, 1996.

[6] D. Hawkins. *Identification of Outliers*. Chapman and Hall, London, 1980.

[7] A. Hinneburg, and D. A. Keim. An Efficient Approach to Cluster in Large Multimedia Databases with Noise. *SIGKDD'98*, New York, NY, pp 58-65, 1998.

[8] W. Jin, A. K. H. Tung and J. Han. Finding Top n Local Outliers in Large Database. *SIGKDD'01*, San Francisco, CA, pp 293-298, 2001.

[9] E. M. Knorr and R. T. Ng. Algorithms for Mining Distance-based Outliers in Large Dataset. *VLDB'98*, New York, NY, pp 392-403, 1998.

[10] E. M. Knorr and R. T. Ng. Finding Intentional Knowledge of Distance-based Outliers. *VLDB'99*, Edinburgh, Scotland, pp 211-222, 1999.

[11] R. Ng and J. Han. Efficient and Effective Clustering Methods for Spatial Data Mining. *VLDB'94*, Santiago, Chile, pp 144-155, 1994.

[12] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient Algorithms for Mining Outliers from Large Data Sets. *SIGMOD'00*, Dallas, Texas, pp 427-438, 2000.

[13] J. Tang, Z. Chen, A. Fu, and D. W. Cheung. Enhancing Effectiveness of Outlier Detections for Low Density Patterns. *PAKDD'02*, Taipei, Taiwan, 2002.

[14] J. Zhang and H. Wang. Detecting Outlying Subspaces for High-dimensional Data: the New Task, Algorithms and Performance. *Knowledge and Information Systems(KAIS)*, Springer-Verlag Publisher, 2006.

[15] J. Zhang, W. Hsu and M. L. Lee. Clustering in Dynamic Spatial Databases. *Journal of Intelligent Information Systems (JIIS)* 24(1): 5-27, Kluwer Academic Publisher, 2005.

[16] C. Zhu, H. Kitagawa and C. Faloutsos. Example-Based Robust Outlier Detection in High Dimensional Datasets. *ICDM'05*, pp 829-832, 2005.

[17] C. Zhu, H. Kitagawa, S. Papadimitriou, and C. Faloutsos. OBE: Outlier by Example. *PAKDD'04*, pp 222-234, Sydney, Australia, 2004.

[18] J. Zhang, M. Lou, T. W. Ling and H. Wang. HOS-Miner: A System for Detecting Outlying Subspaces of High-dimensional Data. *VLDB'04*, pp 1265-1268, Toronto, Canada, 2004.

[19] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An Efficient Data Clustering Method for Very Large Databases. *SIGMOD'96*, Montreal, Canada, pp 103-114, 1996.