Outlier Mining in Large High-Dimensional Data Sets

Fabrizio Angiulli and Clara Pizzuti
ICAR-CNR
c/o DEIS, Università della Calabria
Via Pietro Bucci, 41C
87036 Rende (CS) - Italy
Email: {angiulli, pizzuti}@icar.cnr.it

Abstract

In this paper a new definition of distance-based outlier and an algorithm, called *HilOut*, designed to efficiently detect the top n outliers of a large and high-dimensional data set are proposed. Given an integer k, the weight of a point is defined as the sum of the distances separating it from its k nearest-neighbors. Outlier are those points scoring the largest values of weight. The algorithm *HilOut* makes use of the notion of space-filling curve to linearize the data set, and it consists of two phases. The first phase provides an approximate solution, within a factor $\mathcal{O}(kd^{1+\frac{1}{t}})$, where d is the number of dimensions of the data set and t identifies the L_t metrics of interest, after the execution of at most $d+1$ sorts and scans of the data set, with temporal cost $\mathcal{O}(d^2Nk)$ and spatial cost $\mathcal{O}(Nd)$, where N is the number of points in the data set. The second phase calculates the exact solution with a final scan of temporal cost $\mathcal{O}(N^*Nd)$, where N^* is the number of candidate outliers remained after the first phase. During this phase, the algorithm isolate points candidate to be outliers and reduces this set at each iteration. Thus, if this set becomes of size n, then the algorithm stops reporting the exact solution. Experimental results show that the algorithm always stops, reporting the exact solution, during the first phase after $\overline{d}$ steps, with $\overline{d}$ much less than $d+1$. We present both an in-memory and disk-based implementation of the *HilOut* algorithm and a throughout scaling analysis for real and synthetic data sets showing that the algorithm scales well in both cases.

1 Introduction

Outlier detection is an outstanding data mining task, referred to as *outlier mining*, that has a lot of practical applications in many different domains. Outlier mining can be defined as follows: "Given a set of N data points or objects and the number n of expected outliers, find the top n objects that are considerably dissimilar, exceptional or inconsistent with respect to the remaining data" [12]. Many data mining algorithms consider outliers as noise that must be eliminated because it degrades their predictive accuracy. For example, in classification algorithms mislabelled instances are considered outliers and thus they are removed from the training set to improve the accuracy of the resulting classifier [7]. However, as pointed out in [12], "one person's noise could be another person's signal", thus outliers themselves can be of great interest. Outlier mining can be used in telecom or credit card frauds to detect the atypical usage of telecom services or credit cards, in medical analysis to test abnormal reactions to new medical therapies, in pharmaceutical research, in financial applications, in weather prediction, in marketing and customer segmentations to identify customers spending much more or much less the average customer. Outlier mining actually consists of two subproblems [12]: first define what data is deemed to be exceptional in a given data set, second find an efficient algorithm to obtain such data. Outlier detection methods can be categorized in several approaches, each assumes a specific concept of what an outlier is. Among them, the *distance-based* approach, introduced by Knorr and Ng [15], provides a definition of distance-based outlier relying on the Euclidean distance between two points. This kind of definition, is relevant in a wide range of real-life application domains, as showed in [16].

In this paper we propose a new definition of distance-based outlier and an efficient algorithm, called *HilOut*, designed to detect the top n outliers of a large and high-dimensional data set. Given an application dependent parameter k, the *weight* of a point is defined as the sum of the distances separating it from its k nearest-neighbors. Outlier are thus the points scoring the largest values of weight. The computation of the weights, however, is an expensive task because it involves the calculation of the k nearest neighbors of each data point. To overcome this problem we present a definition of approximate set of outliers. Elements of this set have a weight greater than the weight of true outliers within a small factor. This set of points represents the points candidate to be the true outliers. Thus we give an algorithm consisting of two phases. The first phase provides an approximate solution, within a factor $\mathcal{O}(kd^{1+\frac{1}{t}})$, where d is the number of dimensions of the data set and t identifies the L_t metrics of interest, after executing at most $d+1$ sorts and scans of the data set, with temporal cost $\mathcal{O}(d^2Nk)$ and spatial cost $\mathcal{O}(Nd)$, where N is the number of points in the data set. The algorithm avoids the distance computation of each pair of points because it makes use of the space-filling curves to linearize the data set. We fit the d-dimensional data set $\mathbf{DB}$ in the hypercube $D = [0,1]^d$, then we map D into the interval $I = [0,1]$ by using the *Hilbert space filling curve* and obtain the approximate k nearest neighbors of each point by examining its predecessors and successors on I. The mapping assures that if two points are close in I, they are close in D too, although the reverse in not always true. To limit the loss of nearness, the data set is shifted $d+1$ times along the main diagonal of the hypercube $[0,2]^d$. During each scan the algorithm calculates a lower and an upper bound to the weight of each point and exploits such information to isolate points candidate to belong to the solution set. The number of points candidate to belong to the solution set is sensibly reduced at each scan. Hence, the first phase produces a set of approximate outliers that are candidate to be the true outliers. The second phase calculates the exact solution with a final scan of temporal cost $\mathcal{O}(N^*Nd)$, where N^* is the number of candidate outliers remained after the first phase. Experimental results show that the algorithm always stops, reporting the exact solution, during the first phase after $\overline{d}$ steps, with $\overline{d}$ much less than $d+1$. We present both an in-memory and disk-based implementation of the *HilOut* algorithm and a throughout scaling analysis for real and synthetic data sets showing that the algorithm scales well in both cases.

The rest of the paper is organized as follows. Next section gives an overview of the existing approaches to outlier mining. Section 3 gives definitions and properties necessary to introduce the algorithm and an overview of space filling curves. Section 4 presents the method, provides the complexity analysis and extends the method when the data set does not fit in main memory. In Section 5, finally, experimental results on several data sets are reported.

2 Related Work

The approaches to outlier mining can be classified in supervised-learning based methods, where each example must be labelled as exceptional or not [18, 23], and the unsupervised-learning based ones, where the label is not required. The latter approach is more general because in real situations we do not have such information. In this paper we deal only with unsupervised methods.

Unsupervised-learning based methods for outlier detection can be categorized in several approaches. The first is *statistical-based* and assumes that the given data set has a distribution model. Outliers are those points that satisfies a discordancy test, that is that are significantly larger (or smaller) in relation to the hypothesized distribution [4]. In [30] a Gaussian mixture model to represent the normal behaviors is used and each datum is given a score on the basis of changes in the model. High score indicates high possibility of being an outlier. This approach has been combined in [29] with a supervised-learning based approach to obtain general patterns for outliers.

Deviation-based techniques identify outliers by inspecting the characteristics of objects and consider an object that deviates from these features an outlier [3, 25].

A completely different approach that finds outliers by observing *low dimensional projections* of the search space is presented in [1]. Thus a point is considered an outlier, if it is located in some low density subspace. In order to find the lower dimensional projections presenting abnormally low density, the authors use a *Genetic Algorithm* that quickly find combinations of dimensions in which data is sparse.

Yu et al. [9] introduced *FindOut*, a method based on wavelet transform, that identifies outliers by removing clusters from the original data set. Wavelet transform has also been used in [28] to detect outliers in stochastic processes.

Another category is the *density-based*, presented in [6] where a new notion of local outlier is introduced that measures the degree of an object to be an outlier with respect to the density of the local neighborhood. This degree is called *Local Outlier Factor LOF* and is assigned to each object. The computation of LOFs, however, is expensive and it must be done for each object. To reduce the computational load, Jin et al. in [13] proposed a new method to determine only the top-n local outliers that avoids the computation of LOFs for most objects if $n \ll N$, where N is the data set size.

Distance-based outlier detection has been introduced by Knorr and Ng [15, 16] to overcome the limitations of statistical methods. A *distance-based* outlier is defined as follows: *A point p in a data set is an outlier with respect to parameters k and δ if at least k points in the data set lies greater than distance δ from p.* This definition generalizes the definition of outlier in statistics and it is suitable when the data set does not fit any standard distribution. The authors present two algorithms, one is a nested-loop algorithm that runs in $O(dN^2)$ time and the other one is a cell-based algorithm that is linear with respect to N but exponential in the number of dimensions d. Thus this last method is fast only if $d \leq 4$.

The definition of outlier given by Knorr and Ng, as observed in [22], has a number of benefits, such as being intuitive and computationally feasible for large data sets, but it depends on the

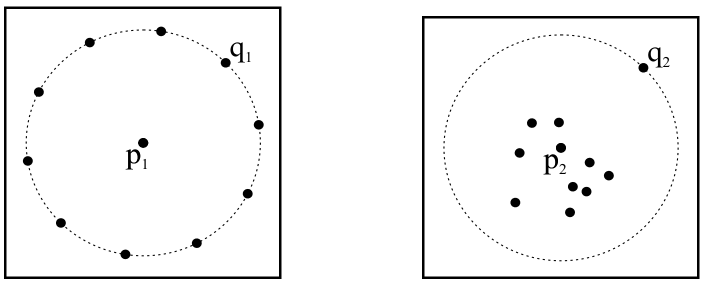


Figure 1: Two points with same D^k values (k=10).

two parameters k and δ and it does not provide a ranking of the outliers. In order to address these drawbacks, Ramaswamy et al. [22] modified the definition of outlier. The new definition of outlier is based on the distance of the k-th nearest neighbor of a point p, denoted with $D^k(p)$, and it is the following: *Given a k and n, a point p is an outlier if no more than n-1 other points in the data set have a higher value for D^k than p.* This means that the top n points having the maximum D^k values are considered outliers. To detect outliers, a partition-based algorithm is presented that first partitions the input points using a clustering algorithm, namely BIRCH [31], then it prunes those partitions that cannot contain outliers. These partitions are determined by computing upper and lower bounds on D^k for points in each partition. For each partition P, lower and upper bounds are also computed and only those partitions having an upper bound greater than the lower bound determined for the n outliers are maintained. A final step finds the outliers considering only the points contained in the candidate partitions. The experiments presented, up to 10 dimensions, show that the method scales well with respect to both data set size and dimensionality.

The authors note that "points with large values for $D^k(p)$ have more sparse neighborhoods and are thus typically stronger outliers than points belonging to dense clusters which will tend to have lower values of $D^k(p)$." However, consider Figure 1. If we set $k = 10$, $D^k(p_1) = D^k(p_2)$, but we can not state that p_1 and p_2 can be considered being outliers at the same way.

In the next section we propose a new definition of outlier that is distance-based but that considers for each point p the sum of the distances from its k nearest neighbors. This sum is called the *weight* of p, $\omega_k(p)$, and it is used to rank the points of the data set. Outliers are those points having the largest values of ω_k. $\omega_k(p)$ is a more accurate measure of how much of an outlier point p is because it takes into account the sparseness of the neighborhood of a point. In the above figure, intuitively, p_2 does not seem to be an outlier, while p_1 can be. Our definition is able to distinguish this kind of situations by giving a higher weight to p_1.

3 Definitions

In this section we present the new definition of outlier, the notion of space-filling curve, and we introduce same further definitions that are necessary to describe our outlier detection algorithm.

3.1 Defining outliers

Let t be a positive number, then the L_t *distance* between two points $p = (p_1, \ldots, p_d)$ and $q = (q_1, \ldots, q_d)$ of $\mathbb{R}^d$ is defined as $\mathrm{d}_t(p, q) = (\sum_{i=1}^{d} |p_i - q_i|^t)^{1/t}$ for $1 \leq t < \infty$, and as $\max_{1 \leq i \leq d} |p_i - q_i|$, for $t = \infty$.

Definition 1 (Weight) Let **DB** be a d-dimensional data set, k a parameter and p a point of **DB**. Then the *weight* of p in **DB** is defined as $\omega_k(p) = \sum_{i=1}^{k} \mathrm{d}_t(p, nn_i(p))$, where $nn_i(p)$ denotes the i-th nearest neighborhood of p in **DB** according to the L_t distance. That is, the weight of a point is the sum of the distances separating that point from its k nearest neighbors.

Intuitively, the notion of weight captures the degree of isolation of a point with respect to its neighbors, higher is its weight, more distant are its neighbors.

Definition 2 (Outlier) Let **DB** be a data set, k and n two parameters, and let p be a point of **DB**. Then p is the n-th outlier with respect to k in **DB**, denoted as $outlier_k^n$, if there are exactly $n-1$ points q in **DB** such that $\omega_k(q) \geq \omega_k(p)$. We denote with Out_k^n the set of the top n outliers of **DB** with respect to k.

Thus, given n, the expected number of outliers in the data set, and an application dependent parameter k, specifying the size of the neighborhood of interest, the **Outlier Detection Problem** consists in finding the n points of the data set scoring the maximum ω_k values.

The computation of the weights is an expensive task because it involves the calculation of k nearest neighbors of each data point. While this problem is well solved in any fixed dimension, requiring $\mathcal{O}(\log N)$ time to perform each search (with appropriate space and preprocessing time bounds) [21], when the dimension d is not fixed, the proposed solutions become impracticable since they have running time logarithmic in N but exponential in d. The lack of efficient algorithms when the dimension is high is known as "curse of dimensionality" [5]. In these cases a simple linear scan of the data set, requiring $\mathcal{O}(N^2 d)$ time, outperforms the proposed solutions.

From what above stated, at the present, when large and high-dimensional data sets are considered, a good algorithm for the solution of the Outlier Detection Problem is the naive nested-loop algorithm which, in order to compute the weight of each point, it must consider the distance from all the points of the data set, thus requiring $\mathcal{O}(N^2 d)$ time. Data mining applications, however, require algorithms that scale near linearly with the size of the data set to be practically applicable. An approach to overcome this problem could be to first find an approximate, but fast, solution, and then obtain the exact solution from the approximate one. This motivate our definition of approximation of a set of outliers.

Definition 3 (Approximation of $\mathbf{Out_k^n}$) Let **DB** be a data set, let $Out^* = \{a_1, \ldots, a_n\}$ be a set of n points of **DB**, with $\omega_k(a_i) \geq \omega_k(a_{i+1})$, for $i = 1, \ldots, n-1$, and let ϵ be a positive real number greater than one. We say that Out^* is an *ϵ-approximation* of Out_k^n, if $\epsilon\omega_k(a_i) \geq \omega_k(outlier_k^i)$, for each $i = 1, \ldots, n$.

In the following sections we give an algorithm that computes an approximate solution within a factor $\mathcal{O}(kd^{1+\frac{1}{t}})$, where t is the L_t metrics of interest, runs in $\mathcal{O}(d^2 Nk)$ time and has spatial cost $\mathcal{O}(Nd)$. The algorithm avoids the distance computation of each pair of points because it makes use of the space-filling curves to linearize the data set. To obtain the k approximate nearest neighbors of each point p it is sufficient to consider its successors and predecessors on the linearized data set. The algorithm produces a set of approximate (with respect to the above definition) outliers that are candidate to be the true outliers. The exact solution can then be obtained from this candidate set at a low cost.

In the next subsection the concept of space-filling curve is recalled.

3.2 Space-filling curves

The concept of *space-filling curve* came out in the 19-th century and is accredited to Peano [24] who, in 1890, proved the existence of a continuous mapping from the interval $I = [0, 1]$ onto the square $Q = [0, 1]^2$. Hilbert in 1891 defined a general procedure to generate an entire class of space-filling curves. He observed that if the interval I can be mapped continuously onto the square Q then, after partitioning I into four congruent subintervals and Q into four congruent sub-squares, each subinterval can be mapped onto one of the sub-squares. Sub-squares are ordered such that each pair of consecutive sub-squares share a common edge. If this process is continued ad infinitum, I and Q are partitioned into 2^{2h} replicas for $h = 1, 2, 3 \ldots$. Figure 2 shows the first three steps of this process. Sub-squares are arranged so that the inclusion relationships and adjacency property are always preserved. In practical applications the partitioning process is terminated after h steps to give an approximation of a space-filling curve of order h. For $h \geq 1$ and $d \geq 2$, let $\mathcal{H}_h^d$ denote the h-th order approximation of a

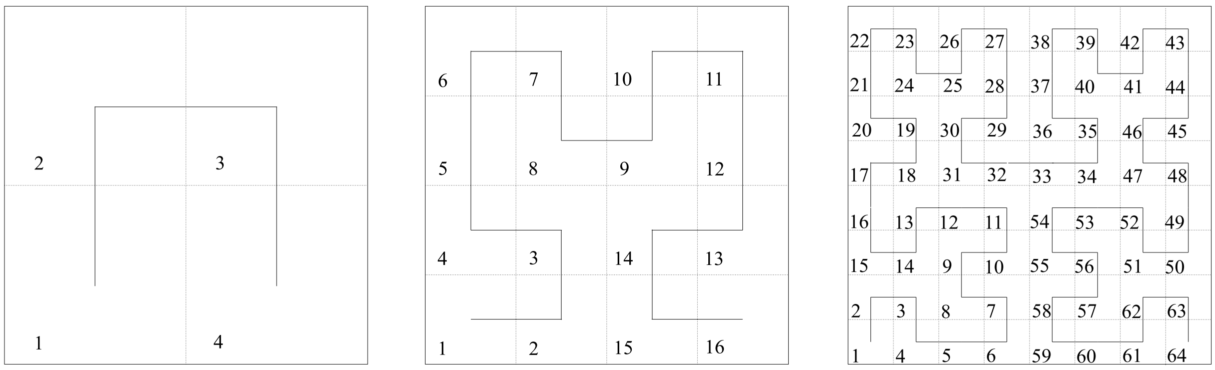


Figure 2: The Hilbert Space-Filling Curve.

d-dimensional Hilbert space-filling curve that maps 2^{hd} subintervals of length $1/2^{hd}$ into 2^{hd} sub-hypercubes whose centre-points are considered as points in a space of finite granularity. The Hilbert curve, thus, passes through every point in a d-dimensional space once and once only in a particular order. This establishes a mapping between values in the interval I and the coordinates of d-dimensional points. Let D be the set $[0,1]^d$ and p a d-dimensional point in D. The inverse image of p under this mapping is called its *Hilbert value* and is denoted by $\mathcal{H}(p)$. Let $\mathbf{DB}$ be a set of points in D. These points can be sorted according to the order in which the curve passes through them. We denote by $\mathcal{H}(\mathbf{DB})$ the set $\{\mathcal{H}(p) \mid p \in \mathbf{DB}\}$ sorted with respect to the order relation induced by the Hilbert curve. Given a point p the predecessor and the successor of p, denoted $\mathcal{H}_{pred}(p)$ and $\mathcal{H}_{succ}(p)$, in $\mathcal{H}(\mathbf{DB})$ are thus the two closest points with respect to the ordering induced by the Hilbert curve. The *m-th* predecessor and successor of p are denoted by $\mathcal{H}_{pred}(p,m)$ and $\mathcal{H}_{succ}(p,m)$. Space filling curves have been studied and used in several fields [10, 11, 14, 2, 20, 26, 27]. A useful property of such a mapping is that if two points from the unit interval I are close then the corresponding images are close too in the hypercube D. The reverse statement, however, is not true because two close points in D can have non-close inverse images in I. This implies that the reduction of dimensionality from d to one can provoke the loss of the property of nearness. In order to preserve the closeness property, approaches based on the translation and/or rotation of the hypercube D have been proposed [19, 26]. Such approaches assure the maintenance of the closeness of two d-dimensional points, within some factor, when they are transformed into one dimensional points. In particular, in [19], the number of shifts depends on the dimension d. Given a data set $\mathbf{DB}$ and the vector $v^{(j)} = (j/(d+1), \ldots, j/(d+1)) \in \mathbb{R}^d$, each point $p \in \mathbf{DB}$ can be translated $d+1$ times along the main diagonal obtaining points $p^j = p + v^{(j)}$, for $j = 0, \ldots, d$. The shifted copies of points thus belong to $[0,2]^d$ and, for each p, $d+1$ Hilbert values in the interval $[0,2]$ can be computed.

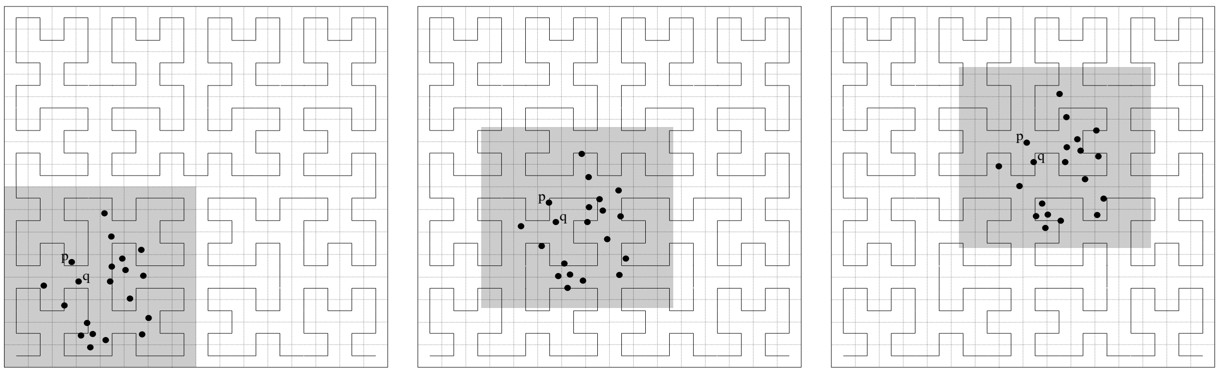


Figure 3: Shifts of a two dimensional data set.

Figure 3 shows the three shifted copies of a two dimensional data set. Note that, in the first shift, points p and q are not close according to the order induced on the data set by the curve, while during the second shift they are close according the the same order. In this paper we make use of this family of shifts to overcome the loss of the nearness property.

3.3 Further definitions and properties

We now give some other definitions that will be used throughout the paper.

Definition 4 An *r-region* is an open ended hypercube in $[0,2)^d$ with side length $r = 2^{1-l}$ having the form $\prod_{i=0}^{d-1}[a_i r, (a_i+1)r)$, where each a_i, $0 \leq i < d$, and l are in $\mathbb{N}$. The *order* of an r-region of side r is the quantity $-\log_2 r$.

Definition 5 Let p and q be two points. We denote by *MinReg*(p,q) the side of smallest r-region containing both p and q. We denote by *MaxReg*(p,q) the side of the greatest r-region containing p but not q.

The functions *MaxReg* and *MinReg* can be calculated in time $\mathcal{O}(d)$ by working on the bit string representation of the values $\mathcal{H}(p)$ and $\mathcal{H}(q)$.

Definition 6 Let p be a point, and let r be the side of an r-region. Then

$$MinDist(p,r) = \min_{i=1}^{d}\{\min\{p_i \bmod r, r-(p_i \bmod r)\}\}$$

where $x \bmod r = x - \lfloor x/r \rfloor r$, and p_i denotes the value of p along the i-th coordinate, is the perpendicular distance from p to the nearest face of the r-region of side r containing p, i.e. a lower bound for the distance between p and a point lying out of the above r-region.

In the definition of *MinDist* we assume that the faces of the r-region lying on the surface of the hypercube $[0,2]^d$ are ignored, i.e. if $p_i < r$ ($2 - r \le p_i$ resp.) then only the term $r - (p_i \bmod r)$ ($p_i \bmod r$ resp.) is taken into account, for each $i = 1, \dots, d$.

Definition 7 Let p be a point, and let r be the side of an r-region. Then

$$MaxDist(p,r) = \begin{cases} \left(\sum_{i=1}^{d} (\max\{p_i \bmod r, r - (p_i \bmod r)\})^t \right)^{\frac{1}{t}} & \text{, for } 1 \le t < \infty \\ \max_{i=1}^{d} \{\max\{p_i \bmod r, r - (p_i \bmod r)\}\} & \text{, for } t = \infty \end{cases}$$

is the distance from p to the furthest vertex of the r-region of side r containing p, i.e. an upper bound for the distance between p and a point lying into the above r-region.

Definition 8 Let p be a point in $\mathbb{R}^d$, and let r be a non negative real. Then the d-dimensional *neighborhood* of p (under the L_t metric) of radius r, written $\mathcal{B}(p,r)$, is the set $\{q \in \mathbb{R}^d \mid \mathrm{d}_t(p,q) \le r\}$.

Definition 9 Let p, q_1, and q_2 be three points. Then

$$BoxRadius(p, q_1, q_2) = MinDist(p, \min\{MaxReg(p, q_1), MaxReg(p, q_2)\})$$

is the radius of the greatest neighborhood of p entirely contained in the greatest r-region containing p but neither q_1 nor q_2.

Lemma 1 *Given a data set* **DB***, a point p of* **DB***, two positive integers a and b, and the set of points*

$$I = \{\mathcal{H}_{pred}(p,a), \dots, \mathcal{H}_{pred}(p,1), \mathcal{H}_{succ}(p,1), \dots, \mathcal{H}_{succ}(p,b)\}$$

let r be $BoxRadius(p, \mathcal{H}_{pred}(p, a-1), \mathcal{H}_{succ}(p, b+1))$ and $S = I \cap \mathcal{B}(p,r)$. Then

1. *The points in S are the true first $|S|$ nearest-neighbors of p in* **DB**
2. $\mathrm{d}_t(p, nn_{|S|+1}(p)) > r$

Proof. First, we note that, for each r-region, the intersection of the Hilbert space-filling curve, with the r-region results in a connected segment of the curve. Hence, to reach the points $\mathcal{H}_{pred}(p, a-1)$ and $\mathcal{H}_{succ}(p, b+1)$ from p following the Hilbert curve, we surely walk through the entire r-region of side r_b containing p. As the distance from p to the nearest face of its r_b-region is r_n, then $\mathcal{B}(p, r_n)$ is entirely contained in that region. It follows that the points in S

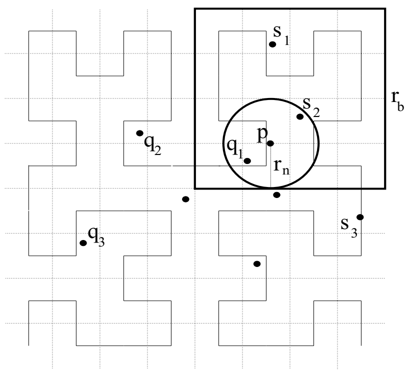


Figure 4: An example of application of Lemma 1.

are all and the only points of **DB** placed at a distance not greater than r_n from p. Obviously, the $(|S| + 1)$-th nearest-neighbor of p has a distance greater than r_n from p. $\square$

The above Lemma allows us to determine, among the $a + b$ points, nearest neighbors of p with respect to the Hilbert order (thus they constitute an approximation of the true closest neighbors), the exact $|S| \leq a + b$ nearest neighbors of p and to establish a lower bound to the distance from p to the $(|S| + 1)$-th nearest neighbor. This result is used in the algorithm to estimate a lower bound to the weight of any point p. Figure 4 shows an example of application of Lemma 1, with the r_b-region, the distance r_b, and $\mathcal{B}(p, r_n)$, for $a = b = 2$.

4 Algorithm

In this section we give the description of the *HilOut* algorithm, which solves the Outlier Detection Problem. The method consists of two phases, the first does at most $d + 1$ sorts and scans of the input data set and guarantees a solution that is an $k\epsilon_d$-approximation of Out^n_k, where $\epsilon_d = \mathcal{O}(d^{1+\frac{1}{t}})$, with a low time complexity cost. The second phase does a single scan of the data set and computes the set Out^n_k.

At each scan *HilOut* computes a lower bound and an upper bound to the weight of each point and it maintains the n greatest lower bound values of weight in a heap. The lowest value in this heap is a lower bound to the weight of the n-th outlier and it is used to detect those points that can be considered candidate outliers.

The upper and lower bound of the weight of each point are computed by exploring the neighborhood of the point according to the Hilbert order. The size of this neighborhood is initially set to $2k$, then it is widened, proportionally to the number of remaining candidate outliers, to obtain a better estimate of the true k nearest neighbors.

At each iteration, as experimental results show, the number of candidate outliers sensibly diminishes. This allows the algorithm to find the exact solution in few steps, in practice after $\overline{d}$ steps with $\overline{d}$ much less than $d+1$.

Before starting with the description of *HilOut*, we introduce the concept of *point feature*.

Definition 10 A *point feature* f is a 7-tuple

$$\langle id, point, hilbert, level, ubound, lbound, nn \rangle$$

where

- *id* is an unique identifier associated to f
- *point* is a point in $[0,2)^d$
- *hilbert* is the Hilbert value associated to *point* in the h-th order approximation of the d-dimensional Hilbert space-filling curve mapping the hypercube $[0,2)^d$ into the integer set $[0, 2^{hd})$
- *level* is the order of the smallest r-region containing both *point* and its successor in **DB** with respect to the Hilbert order
- *ubound* and *lbound* are respectively an upper and a lower bound to the weight of *point* in **DB**
- *nn* is a set of at most k pairs $(id', dist)$, where id' is the identifier associated to another point feature f', and *dist* is the distance between the point stored in f and the point stored in f'

If *nn* is not empty, we say that f is an *extended point feature*.

In the following, with the notation *f.id*, *f.point*, *f.hilbert*, *f.level*, *f.ubound*, *f.lbound* and *f.nn* we denote respectively the *id*, *point*, *hilbert*, *level*, *ubound*, *lbound* and *nn* value of the point feature f.

We recall that with $v^{(j)}$ we denote the d-dimensional point $(j/(d+1), \ldots, j/(d+1))$.

The algorithm *HilOut*, reported in Figure 5, receives as input a data set **DB** of N points in the hypercube $[0,1]^d$, the number n of top outliers to find and the number k of neighbors to consider.

The data structures employed by the algorithm are the two heaps OUT and WLB, the set TOP, and the list of point features PF:

- OUT and WLB are two heaps of n point features. At the end of each iteration, the features stored in OUT are those with the n greatest values of the field *ubound*, while the features stored in WLB are those with the n greatest values of *lbound*
- TOP is a set of at most $2n$ point features which is set to the union of the features stored in OUT and WLB at the end of the previous iteration
- PF is a list of point features. In the following, with the notation PF_i we mean the i-th element of the list PF

First, the algorithm builds the list PF associated to the input data set, i.e. for each point p of **DB** a point feature f with its own $f.id$ value, $f.point = p$, $f.ubound = \infty$, $f.level$ and $f.lbound$ set to 0, and $f.nn = \emptyset$, is inserted in PF, and initializes the set TOP and the global variables ω^*, N^*, and n^*:

- ω^* is a lower bound to the weight of the $outlier^n_k$ in **DB**. This value, initially set to 0, is then updated in the procedure *Scan*
- N^* is the number of point features f of PF such that $f.ubound \geq \omega^*$. The points whose point feature satisfies the above relation are called *candidate outliers* because the upper bound to their weight is greater than the current lower bound ω^*. This value is updated in the procedure *Hilbert*
- n^* is the number of true outliers in the heap OUT. It is updated in the procedure *TrueOutliers* and it is equal to $|\{f \in OUT : f.lbound = f.ubound \wedge f.ubound \geq \omega^*\}|$

The main cycle, consists of at most $d + 1$ steps. We explain the single operations performed during each step of this cycle.

Hilbert. The *Hilbert* procedure calculates the value $\mathcal{H}(PF_i.point + v^{(j)})$ of each point feature PF_i of PF, where $j \in \{0, \ldots, d\}$ identifies the current main iteration, places this value in $PF_i.hilbert$, and sorts the point features in the list PF using as order key the values $PF_i.hilbert$. Thus it performs the Hilbert mapping of a shifted version of the input data set. It is straightforward to note that the shift operation does not alter the mutual distances between the points in PF. As $v^{(0)}$ is the zero vector, at the first step ($j = 0$) no shift is performed. Thus during this step we work on the original data set. After sorting, the procedure *Hilbert* updates the value of the field *level* of each point feature. In particular, the value $PF_i.level$ is set to the order of the smallest r-region containing both $PF_i.point$ and $PF_{i+1}.point$, i.e. to $MinReg(PF_i.point, PF_{i+1}.point)$, for each $i = 1, \ldots, N-1$. For example, consider figure 6

```
Algorithm HilOut (DB, $n$, $k$)
begin
    Initialize($PF$, DB);
    (* First Phase *)
    $TOP := \emptyset$;
    $N^* := N$; $n^* := 0$; $\omega^* := 0$;
    $j := 0$;
    while ($j \leq d$) and ($n^* < n$) do begin
        Initialize($OUT$);
        Initialize($WLB$);
        Hilbert($v^{(j)}$);
        Scan($v^{(j)}$, $\frac{kN}{N^*}$);
        TrueOutliers($OUT$);
        $TOP := OUT \cup WLB$;
        $j := j + 1$;
    end;
    (* Second Phase *)
    if $n^* < n$ then Scan($v^{(d)}$, $N$);
    return $OUT$;
end.
```

Figure 5: The algorithm *HilOut* and the procedure *Scan*

where seven points in the square $[0,1]^2$ are consecutively labelled with respect to the Hilbert order. Figure 6 (b) highlights the smallest r-region containing the two points 5 and 6, while Figure 6 (c) that containing the two points 2 and 3. The values of the levels associated with the points 5 and 2 are thus three and one because the order of corresponding r-regions are $-\log_2 2^{1-4} = 3$ and $-\log_2 2^{1-2} = 1$ respectively. On the contrary, the smallest r-region containing points 1 and 2 is all the unit square.

Scan. The procedure *Scan* is reported in Figure 7. This procedure performs a sequential scan of the list PF by considering only those features that have a weight upper bound not less than ω^*, the lower bound to the weight of $outlier_k^n$ of **DB**. These features are those candidate to be outliers, the others are simply skipped.

If the value $PF_i.lbound$ is equal to $F_i.ubound$, then this is the true weight of $PF_i.point$ in **DB**. Otherwise $PF_i.ubound$ is an upper bound for the value $\omega_k(PF_i.point)$ and it could be improved. For this purpose the function *FastUpperBound* calculates a novel upper bound ω to the weight of $PF_i.point$, given by $k \times MaxDist(PF_i.point, 2^{-level_0})$, by examining k points among its successors and predecessors to find $level_0$, the order of the smallest r-region containing both $PF_i.point$ and other k neighbors. If ω is less than ω^*, no further elaboration is required, as in

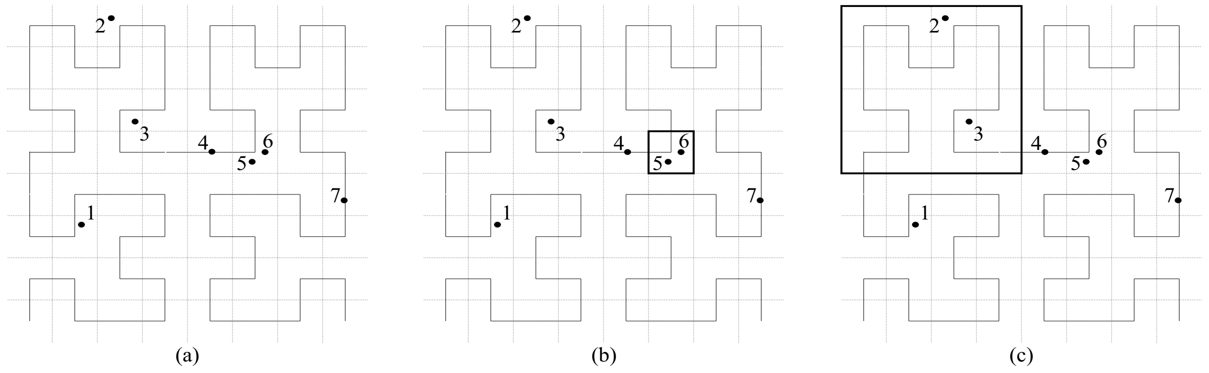


Figure 6: The *level* field semantics

this case the point is not a candidate outlier.

Otherwise the procedure *InnerScan* returns a new lower bound *newlb* and a new upper bound *newub* for the weight of PF_i*.point* (see the description of *InnerScan* below for details regarding the calculation of these bounds).

If *newlb* is greater than PF_i*.lbound* then a better lower bound for the weight of PF_i*.point* is available, and the field *lbound*, is updated. Same considerations hold for the value PF_i*.ubound*. Next, the heaps OUT and WLB process PF_i. That is, if PF_i*.ubound* is greater than the smallest upper (lower resp.) bound *f.ubound* (*f.lbound* resp.) stored in OUT (WLB resp.), than the point feature f stored in OUT (WLB resp.) is replaced with PF_i. Finally, the lower bound ω^* to the weight of the n-th outlier is updated if a greater lower bound has been computed.

InnerScan. This procedure takes into account the set of points

$$PF_a.point, \ldots, PF_{i-1}.point, PF_{i+1}.point, \ldots, F_b.point$$

i.e. the points whose Hilbert value lies in a one dimensional neighborhood of the integer value PF_i*.hilbert*. The maximum size allowed for the above neighborhood is stored in the input parameter *maxcount*. In particular, if PF_i belongs to TOP, i.e. the point is a candidate to be one of the n top outliers we are searching for, then the size $b - a$ of the above neighborhood is at most N, the size of the entire data set, otherwise this size is at most $2k_0$.

We note that the parameter k_0, that is the number of neighbors to consider on the above interval, of the procedure *Scan* is set to kN/N^*, i.e. it is inversely proportional to the number N^* of candidate outliers at the beginning of the current main iteration. This allows the algorithm to analyze further the remaining candidate outliers, maintaining at the same time the number of distance computations performed in each iteration constant.

```
procedure $Scan(v,\ k_0)$;
begin
    for $i := 1$ to $N$ do if $(PF_i.ubound \geq \omega^*)$ then begin
        if $(PF_i.lbound < PF_i.ubound)$ then begin
            $\omega := FastUpperBound(i)$;
            if $(\omega < \omega^*)$ then $F_i.ubound := \omega$ else begin
                $maxcount := \min(2k_0, N)$;
                if $(PF_i \in TOP)$ then $maxcount := N$;
                $InnerScan(i,\ maxcount,\ v,\ PF_i.nn,\ newlb,\ newub)$;
                if $(newlb > PF_i.lbound)$ then
                    $PF_i.lbound := newlb$;
                if $(newub < PF_i.ubound)$ then
                    $PF_i.ubound := newub$;
            end;
        end;
        $Update(OUT,\ PF_i)$;
        $Update(WLB,\ PF_i)$;
        $\omega^* := \max(\omega^*, Min(WLB))$;
    end;
end; { $Scan$ }
```

Figure 7: The algorithm *HilOut* and the procedure *Scan*

This procedure manages the set NN of at most k pairs $(id, dist)$, where id is the identifier of a point feature f and $dist$ is the distance between the current point $PF_i.point$ and the point $f.point$.

The variable *levela* (*levelb* respectively), initialized to the order h of the approximation of the space filling curve, represents the minimum among $PF_{a-1}.level$, …, $PF_{i-1}.level$ ($F_i.level$, …, $F_b.level$ resp.) while *level* represents the maximum between *levela* and *levelb*. Thus *level*+1 is the order of the greatest entirely explored r-region (having side $r = 2^{-(level+1)}$) containing $PF_i.point$.

The values a and b are initially set to i. Then, at each iteration of *InnerScan*, the former is decreased or the latter is increased, until a stop condition occurs or their difference exceeds the maximum size allowed. In particular, during each iteration, if $PF_{a-1}.level$ is greater than $PF_b.level$ then a is decreased, else b is increased. This enforces the algorithm to entirely explore the current r-region, having order *level*, before starting the exploration of the surrounding r-region, having order $level - 1$.

The distances between the point $PF_i.point$ and the points of the above defined set are stored in NN by the procedure *Insert*. In particular $Insert(NN, id, dist)$ works as follows: provided that the pair $(id, dist)$ is not already present in NN, if NN contains less then k elements then

```
**procedure** $\mathit{InnerScan}(i,\ \mathit{maxcount},\ v;\ $ **var** $NN,\ \mathit{newlb},\ \mathit{newub});$
**begin**
    $p := PF_i.\mathit{point};\ \mathit{Initialize}(NN);\ a := i;\ b := i;$
    $\mathit{levela} := h;\ \mathit{levelb} := h;\ \mathit{level} := h;\ \mathit{count} := 0;\ \mathit{stop} :=$ **false**;
    **while** $(\mathit{count} < \mathit{maxcount})$ **and** (**not** $\mathit{stop}$) **do begin**
        $\mathit{count} := \mathit{count} + 1;$
        **if** $(PF_{a-1}.\mathit{level} > PF_b.\mathit{level})$ **then begin**
            $a := a - 1;\ \mathit{levela} := \min(\mathit{levela}, PF_a.\mathit{level});\ c := a;$
        **end else begin**
            $\mathit{levelb} := \min(\mathit{levelb}, PF_b.\mathit{level});\ b := b + 1;\ c := b;$
        **end**;
        $\mathit{Insert}(NN,\ PF_c.\mathit{id},\ \mathrm{d}_t(p, PF_c.\mathit{point}));$
        **if** $(\mathit{Size}(NN) = k)$ **then begin**
            **if** $(\mathit{Sum}(NN) < \omega^*)$ **then** $\mathit{stop} :=$ **true**
            **else if** $(\max(\mathit{levela}, \mathit{levelb}) < \mathit{level})$ **then begin**
                $\mathit{level} := \max(\mathit{levela}, \mathit{levelb});\ \delta := \mathit{MinDist}(p, 2^{-(\mathit{level}+1)});$
                **if** $(\delta \geq \mathit{Max}(NN))$ **then** $\mathit{stop} :=$ **true**;
            **end**;
        **end**;
    **end**;
    $r := \mathit{BoxRadius}(p + v, PF_{a-1}.\mathit{point} + v, PF_{b+1}.\mathit{point} + v);$
    $\mathit{newlb} := \mathit{SumLt}(NN,\ r);$
    $\mathit{newub} := \mathit{Sum}(NN);$
**end**; { $\mathit{InnerScan}$ }
```

Figure 8: The procedure *InnerScan*

the pair $(id, dist)$ is inserted in NN, otherwise if $dist$ is less than the smallest distance stored in a pair of NN then this pair is replaced with the pair $(id, dist)$.

The procedure *InnerScan* stops in two cases. The first case occurs when the value $Sum(NN)$ is less than ω^*, where $Sum(NN)$ denotes the sum of the distances stored in each pair of NN, i.e. when the upper bound to the weight of $PF_i.point$ just determined is less than the lower bound to the weight of the $outlier_k^n$ of $\mathbf{DB}$. This means that $PF_i.point$ is not an outlier. The second case occurs when the value of $level$ decreases and the distance between $PF_i.point$ and the nearest face of its $2^{-(level+1)}$-region exceeds the value $Max(NN)$, i.e. the distance between $PF_i.point$ and its k-th nearest neighbor in $\mathbf{DB}$. This means that we already explored the r-region containing both $PF_i.point$ and its k nearest neighbors.

At the end of the procedure *InnerScan*, the function *BoxRadius* calculates the radius r of the greatest entirely explored neighborhood of $PF_i.point$. This value can be obtained by using lemma 1, more simply by exploiting the values $levela$, $levelb$, $PF_{a-1}.level$ and $PF_b.level$, i.e. as

$$2^{-\max(\min(levela, PF_{a-1}.level), \min(levelb, PF_b.level))}$$

Finally, $newlb$ is set to the sum of the distances stored in NN that are less or equal than r while $newub$ is set to the sum of all the distances stored in NN.

The main cycle of the algorithm *HilOut* stops when $n^* = n$, i.e. when the heap OUT is equal to the set of top n outliers, or after $d+1$ iterations. At the end of the first phase, the heap OUT contains a $k\epsilon_d$-approximation of Out_k^n. Finally, if $n^* < n$, that is if the number of true outliers found by the algorithm is not n, then a final scan computes the exact solution. During this final scan the maximum size of the one dimensional neighborhood to consider for each remained candidate outlier is N, that is the entire data set. This terminates the description of the algorithm.

To conclude, we distinguish between two versions of the above described algorithm:

- **nn-HilOut:** this version of *HilOut* uses extended point features, i.e. the nearest neighbors of each point, determined in the procedure *InnerScan*, are stored in its associated point feature and then reused in the following iterations.
- **no-HilOut:** this version uses point features with the field nn always set to $\emptyset$, i.e. the nearest point determined during each iteration are discarded after their calculation.

The former version of the algorithm has extra memory requirements over the latter version, but in general we expect that *nn-HilOut* presents an improved pruning ability.

Next we state the complexity of the algorithm.

4.1 Complexity analysis

To state the complexity of the *HilOut* algorithm, we first consider the procedures *Scan* and *InnerScan*. The function *FastUpperBound* requires $\mathcal{O}(k+d)$ time, i.e. $\mathcal{O}(k)$ time to find the smallest r-region, having order *level*, including both the point $PF_i.point$ and k others points of **DB**, and $\mathcal{O}(d)$ time to calculate $MaxDist(PF_i.point, 2^{-level})$.
Each iteration of *InnerScan* runs in time $\mathcal{O}(d+\log k)$.

Indeed the distance between two points can be computed in time $\mathcal{O}(d)$, while the set NN can be updated in time $\mathcal{O}(\log k)$, provided that it is stored in a suitable data structure.

Furthermore, the stop condition can be verified in time $\mathcal{O}(d)$, corresponding to the cost of the function *MinDist* (the actual value of both $Sum(NN)$ and $Max(NN)$ can be maintained, with no additional insertion cost, in the data structure associated to NN).

We note that there are at most $2n$ point features for which this cycle is executed at most N times, and at most $N-n$ features for which the same cycle is executed at most $2k$ times.

The functions *BoxRadius* and *SumLt* at the end of *InnerScan*, which require time $\mathcal{O}(d)$ and $\mathcal{O}(k)$ respectively, and the procedures *Update* at the end of *Scan*, which require $\mathcal{O}(\log n)$ time, are executed at most N times.

Summarizing, the temporal cost of *Scan* is

$$\mathcal{O}\left(\underbrace{N(k+d)}_{\textit{FastUpperBound}} + \underbrace{N(d+k+\log n)}_{\textit{BoxRadius} + \textit{SumLt} + \textit{Update}} + \underbrace{N(n+k)(d+\log k)}_{\text{cycle of } \textit{InnerScan}} \right)$$

i.e. $\mathcal{O}(N(n+k)(d+\log k))$. The procedure *Hilbert* runs in time $\mathcal{O}(dN\log N)$, hence the time complexity of the first phase of the algorithm is $dN(d\log N + (n+k)(d+\log k))$.

Without loss of generality, if we assume that $\mathcal{O}(n) = \mathcal{O}(k)$, $k \geq \log N$ and $d \geq \log k$, then the cost of the first phase of the algorithm can be simplified in $\mathcal{O}(d^2Nk)$ or equivalently in $\mathcal{O}(d^2Nn)$. Considered that the naive nested-loop algorithm has time complexity $\mathcal{O}(N(\log n + N(d+\log k)))$, the algorithm is particularly suitable in all the applications in which the number of points N overcomes the product dk or dn. As an example, if we search for the top 100 outliers with respect to $k = 100$ in a one hundred dimensional data set containing one million of points, we expect to obtain the approximate solution with time savings of at least two order of magnitude with respect to the naive approach.

Finally, let N^* be the number of candidate outliers at the end of the first phase. Then the time complexity of the second phase is $N^*(\log n + N(d+\log k))$. We expect that $N^* \ll N$ at the end of the first phase. When this condition occurs, the second phase of the algorithm reduces to a single scan of the data set.

As regard the space complexity analysis, assuming that h is a constant and that the space required to stored a floating point number is constant, then the *nn-HilOut* algorithm requires $\mathcal{O}(N(d + k \log N))$ space, while the *no-HilOut* algorithm requires $\mathcal{O}(dN)$ space, and we note that the size of the input data set is $\mathcal{O}(dN)$.

4.2 Approximation error

Now we show that the solution provided by the first phase of the *HilOut* algorithm is within $kd^{1+\frac{1}{t}}$-approximation of the set Out_k^n. The following lemma is from [8].

Lemma 2 *Suppose d is even. Then, for any point $p \in \mathbb{R}^d$ and $r = 2^{-l}$ $(l \in \mathbb{N})$, there exists $j \in \{0, \ldots, d\}$ such that $p + v^{(j)}$ is $\left(\frac{1}{2d+2}\right)$-central in its r-region.*

The lemma states that if we shift a point p of $\mathbb{R}^d$ at most $d+1$ times in a particular manner, i.e. if we consider the set of points $p+v^{(0)}, \ldots, p+v^{(d)}$, than, in at least one of these shifts, this point must become sufficiently central in an r-region, for each admissible value of r.

We denote by ϵ_d the value $2d^{\frac{1}{t}}(2d+1)$.

Lemma 3 *Let f be a point feature of PF such that $f^*.ubound \geq \omega^*$, where f^* denotes the value of f at the end of the algorithm. Then $f^*.ubound \leq k\epsilon_d\omega_k(f.point)$.*

Proof. Let δ_k be $\mathrm{d}_t(f.point, nn_k(f.point))$. From Lemma 2 it follows that there exists an r-region of side $\frac{r}{4d+4} \leq \delta_k < \frac{r}{2d+2}$ (this inequality defines an unique r-region) and an integer $j \in \{0, \ldots, d\}$ such that $f.point^{(j)} = f.point + v^{(j)}$ is $\frac{1}{2d+2}$-central in the r-region. This implies that the distance δ from $f.point^{(j)}$ and each point belonging to its r-region is at most $d^{\frac{1}{t}}\left(r - \frac{r}{2d+2}\right)$ i.e. $d^{\frac{1}{t}}\frac{2d+1}{2d+2}r$. We note that $f.point^{(j)}$ and its true first k nearest neighbors in the shifted version of **DB**, $nn_1(f.point^{(j)})$, ..., $nn_k(f.point^{(j)})$, belong to the r-region.

As $f^*.ubound \geq \omega^*$ then this condition is satisfied during the overall execution of the algorithm, thus the point feature f is always processed by the procedure *Scan*. Consider the j-th main iteration of the algorithm. Let i be the position occupied by the point feature f in the list PF during this iteration.

If $PF_i.lbound$ equals $PF_i.ubound$, than this value is certainly equal to $\omega_k(PF_i.point)$. Otherwise, we show that $PF_i.ubound$ is less or equal then $k\delta$ when the point feature PF_i is considered.

Assume that PF_i.*ubound* is set to *FastUpperBound*(i). Let *level* be the order of the smallest region containing both PF_i.*point* and at least other k points of **DB**, clearly $2^{-level} \leq r$. Thus, *MaxDist*$(PF_i.point, 2^{-level}) \leq \delta$ implies that $PF_i.ubound \leq k\delta$.

Now assume that PF_i.*ubound* is updated after the procedure *InnerScan* with the value *newub*. By absurd, suppose that the condition $NN.ubound \leq k\delta$ is not satisfied. Then, the set of the first k nearest point of PF_i.*point* among $PF_a.point, \ldots, PF_{i-1}.point, PF_{i+1}.point, \ldots, PF_b.point$ must contain a point lying out of the r-region above defined. But this implies that this r-region contains less that $k+1$ points, a contradiction.

Finally, as the value of f.*ubound* cannot increase in the following iterations, then $f^*.ubound \leq k\delta \leq kd^{\frac{1}{t}}\frac{2d+1}{2d+2}r \leq kd^{\frac{1}{t}}\frac{2d+1}{2d+2}(4d+4)\delta_k \leq k\epsilon_d\delta_k$. Since $\omega_k(f.point) \geq \delta_k$, finally $f^*.ubound \leq k\epsilon_d\omega_k(f.point)$. $\square$

Theorem 1 *Let OUT* denote the value of the heap OUT at the end of the first phase of the algorithm and let Out* be the set $\{f.point \mid f \in OUT^*\}$. Then Out* is a $k\epsilon_d$-approximation of Out_k^n.*

Proof. Let Out^* be $\{a_1, \ldots, a_n\}$, let f_i be the point feature associated to a_i, and let f_i^* the value of this point feature at the end of the first phase, for $i = 1, \ldots, n$. Without loss of generality, assume that $f_i^*.ubound \geq f_{i+1}^*.ubound$, for $i = 1, \ldots, n-1$. As $f_i.ubound$ is an upper bound to the weight of a_i, it must be the case that $f_i^*.ubound \geq \omega_k(outlier_k^i)$, for $i = 1, \ldots, n$. It follows from Lemma 3 that $k\epsilon_d\omega_k(a_i) \geq f_i^*.ubound \geq \omega_k(outlier_k^i)$, for $i = 1, \ldots, n$. Let π be a permutation of $\{1, \ldots, n\}$ such that $\omega_k(a_{\pi(1)}) \geq \ldots \geq \omega_k(a_{\pi(n)})$. Now we show that $k\epsilon_d\omega_k(a_{\pi(i)}) \geq \omega_k(outlier_k^i)$, for $i = 1, \ldots, n$. For each $i = 1, \ldots, n$ we have two possibilities: (a) if $\pi(i) < i$ then $k\epsilon_d\omega_k(a_{\pi(i)}) \geq k\epsilon_d\omega_k(outlier_k^{\pi(i)}) \geq \omega_k(outlier_k^i)$; (b) if $\pi(i) \geq i$ then there exists $j \in \{1, \ldots, i\}$ such that $\omega_k(a_{\pi(i)}) \geq \omega_k(a_j)$, hence $k\epsilon_d\omega_k(a_{\pi(i)}) \geq k\epsilon_d\omega_k(a_j) \geq \omega_k(outlier_k^j) \geq \omega_k(outlier_k^i)$. Thus Out^* is a $k\epsilon_d$-approximation of Out_k^n.

$\square$

4.3 Disk-based Algorithm

We described the algorithm *HilOut* assuming that it works with main memory resident data sets. Now we show how the in-memory algorithm can be adapted to manage efficiently disk-resident data sets. Basically, the disk-based implementation of *HilOut* has the same structure of its memory-based counterpart.

The main difference is that the list PF is disk-resident, stored in a file of point features. In particular, the disk-based algorithm manages two files of point features, called F_{in} and F_{out}, and has an additional input parameter BUF, that is the size (in bytes) of the main memory buffer.

First, the procedure *Initialize* creates the file F_{in} with the appropriate values, and with the field $f.hilbert$ of each record f set to $\mathcal{H}(f.point)$.

The procedure *Hilbert* is substituted by the procedure *Sort*, performing an external sort of the file F_{in} and producing the file F_{out} ordered with respect to the field *hilbert*. We used the polyphase merge sort with replacement selection to establish initial runs [17] to perform the external sort. This procedure requires the number FIL of auxiliary files allowed and the size BUF of the main memory buffer. After the sort, F_{in} is set to the empty file.

The procedure *Scan* (and hence *InnerScan*) performs a sequential scan of the file F_{out} working on a circular buffer of size BUF containing a contiguous portion of the file. We have the following differences with the in-memory implementation:

- After a record is updated (i.e. at the end of each iteration of *Scan*), it is appended to the file F_{in} with the field $f.hilbert$ set to $\mathcal{H}(f.point + v^{(j+1)})$, where j denotes the current main iteration of the algorithm
- The maximum value allowed for the parameter k_0 is limited by the number of records (point features) fitting in the buffer of size BUF
- The records of the set TOP are maintained in main memory during the entire execution of *Scan*, compared with the entire data set, and flushed at the end of the overall scan in the appropriate position of the file F_{in}

As for the second phase of the algorithm, this is substituted by a semi-naive nested-loop algorithm. In practice, the records associated with the remaining candidate outliers are stored in the main memory buffer and compared with the entire data set until their upper bound is greater than ω^*. The heaps OUT and WLB are updated at the end of the scan. If the remained candidate outliers do not fit into the buffer, then multiple scans of the feature file are needed.

When the *nn-HilOut* version of the algorithm is considered, to save space and speed up the external sort step, the additional boolean field *extended* is added to every record. This field specifies the size of the record. Indeed, $f.extended$ set to 0 means that the record f does not contain the field *nn*, while $f.extended$ set to 1 means that the field *nn* is present in f. Thus we have records of variable length. Only records associated with candidate outliers have their field *extended* set to 1. This field is managed as follows:

- When a record f is appended to the file F_{in} at the end of each iteration of *Scan*, the field *nn* is added provided that $f.ubound \geq \omega^*$

- The procedure *Sort* must support records of variable length. Moreover, it is modified so that when it builds the file F_{out} by sorting the file F_{in}, it discharges the fields *nn* of the records f having $f.ubound < \omega^*$ (we note that this condition could not be satisfied when the record f is appended to F_{in}, as ω^* can decrease in the following iterations of *Scan*)

We will see in the experimental results section, that when the disk-based implementation of the *HilOut* algorithm is considered, the extra time needed to *no-HilOut* to prune points from the data set, is partially balanced by the lower time required to perform the external sort of the feature file w.r.t. the *nn-HilOut*.

5 Experimental Results

In this section we present a throughout scaling analysis of the *HilOut* algorithm on large high-dimensional data sets, both real, up to about $275,000$ points in the 60-dimensional space, an synthetic, up to $500,000$ points in the 128-dimensional space.

We implemented the algorithm using the C programming language on a Pentium III 800MHz based machine having 512Mb of main memory[1][2]. We used a 32 bit floating-point type to represent the coordinates of the points and the distances.

Real data sets. We tested the *HilOut* algorithm on the following real data sets: *Landsat* ($d = 60$, $N = 275,465$), *ColorHistogram* ($d = 32$, $N = 68,040$), *CoocTexture* ($d = 16$, $N = 68,040$), and *ColorMoments* ($d = 9$, $N = 68,040$). These data sets represent collections of real images. The points of *ColorHistogram*, *CoocTexture* and *ColorMoments* are image features extracted from a Corel image collection[3], while the points of *Landsat* are normalized feature vectors associated to tiles of a collection of large aerial photos[4].

We searched for the top $n \in \{1, 10, 100, 500\}$ outliers for $k \in \{10, 100, 500\}$ under the L_2 metric ($t = 2$). We used the 2nd order approximation of the d-dimensional Hilbert curve to map the hypercube $[0, 2)^d$ onto the set of integers $[0, 2^{2d})$.

It is worth to note that, in all the experiments considered, the algorithm *HilOut* terminates reporting the exact solution after executing a number of iterations much less than $d + 1$. Thus, we experimentally found that in practice the algorithm behaves as an exact algorithm without

[1]The operating system of the computer is Microsoft Windows XP Professional

[2]We used Microsoft Visual C++ as developing environment

[3]See `http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeature.html` for more information

[4]See `http://vision.ece.ucsb.edu/datasets/index.html` for a detailed description

the need of the second phase.

To give an idea of the time savings obtainable with our algorithm with respect to the nested-loop algorithm, we note that, for example, the latter method required, working in main memory, about 46 hours to compute the top $n = 100$ outliers for $k = 100$ of the *Landsat* data set, while the disk-based *no-HilOut* algorithm required less than 1300 seconds to perform the same computation.

Figures 9 and 10 show the result of the above described experiments when we used the disk-based implementation of $HilOut$. Solid lines are relative to the *nn-HilOut* version of the algorithm, while dashed lines to the *no-HilOut* version. We set the buffer size BUF to 64MB in all the experiments.

Figures 9 (a) and (d), and 10 (a) and (d), show the execution times obtained varying the number k of neighbors to consider from 10 to 500, while Figures 9 (b) and (e), and 10 (b) and (e), show the execution times obtained varying the number n of top outliers to consider from 1 to 500. These curves show that the *no-HilOut* version performs better than the *nn-HilOut* on these data sets.

Figures 9 (c) and (f), and 10 (c) and (f), report, in logarithmic scale, the number of candidate outliers at the beginning of each iteration of the algorithm, when $n = 100$ and for $k \in \{10, 100, 500\}$. These curves show that, at each iteration, the algorithm is able to discharge from the set of the candidate outliers a considerable fraction of the whole data set, and that the pruning ability increases with k. Moreover, the same curves show that the algorithm terminates performing less than $d + 1$ iterations.

In general, we expect that the extra time required by *nn-HilOut* to manage the extended feature file will be repayed by an improved pruning ability of the algorithm, i.e. we expect that *nn-HilOut* performs a smaller number of iterations than the *no-HilOut* version. In almost the experiments considered the previous statement is true. Nevertheless, the *no-HilOut* algorithm scales better than *nn-HilOut* algorithm in these experiments, because the number of iterations performed by the two versions of *HilOut* are nearly identical. Thus, when dealing with real data sets, the *no-HilOut* version appears to be superior to the *nn-HilOut*.

Synthetic data sets. To test the algorithm on synthetic data, we used two families of synthetic data sets called *Gaussian* and *Clusters*.

A data set of the *Gaussian* family is composed by points generated from a normal distribution having standard deviation 1 and scaled to fit into the unit hypercube.

A data set of the *Clusters* family is composed by 10 hyper-spherical clusters, formed by the same number of points generated from a normal distribution with standard deviation 1, having diameter 0.05 and equally spaced along the main diagonal of the unit hypercube. Each cluster is surrounded by 10 equally spaced outliers lying on a circumference of radius 0.1 and center in the cluster center.

The data sets of the same family differs only for their size N and for their dimensionality d. A *Gaussian* data set represents a single cluster while a *Clusters* data set is composed by a collection of well-separated clusters. Figures 11 (a) and (b) show the two dimensional *Gaussian* and *Clusters* data sets together with their top 100 outliers for $k = 100$, with $N = 10,000$ points.

We studied the behavior of the algorithm when the dimensionality d and the size N of the data set, the number n of top outliers we are searching for, and the number k of nearest neighbors to consider are varied. In particular, we considered $d \in \{16, 32, 64, 128\}$, $N \in \{50 \cdot 10^3, 100 \cdot 10^3, 200 \cdot 10^3, 500 \cdot 10^3\}$, $n \in \{1, 10, 100, 1000\}$, $k \in \{10, 100, 1000\}$ and the metrics L_2 ($t = 2$). We also studied how the number of candidate outliers decreases during the execution of the algorithm. We set $h = 2$ in the experiments performed on the *Gaussian* data sets and $h = 4$ in the experiments performed on the *Clusters* data sets.

Analogously to what happened for the real data sets, also in the case of the synthetic data sets in all the experiments considered the algorithm terminated with the exact solution after executing a number of iterations much less than $d + 1$.

Figures 12 and 13 show the result of the above described experiments when we used the disk-based implementation of *HilOut*. Solid lines are relative to the *nn-HilOut* version of the algorithm, while dashed lines to the *no-HilOut* version. We set the buffer size BUF to 64MB in all the experiments.

Figures 12 (a) and 13 (a) show, in logarithmic scale, the execution times obtained varying the size N of the data set from $50 \cdot 10^3$ to $500 \cdot 10^3$ for various values of d, and for $n, k = 100$. Figures 12 (d) and 13 (d) show, in logarithmic scale, the execution times obtained varying the dimensionality d of the data sets from $d = 16$ to $d = 128$ for various values of N, and for $n, k = 100$. The *nn-HilOut* algorithm scales well in all cases, while the performance of the *no-HilOut* algorithm only deteriorates on the *Gaussian* data set for $d = 128$ and $N = 500,000$. This is due to the fact that the *Gaussian* data set becomes more and more “sparse” as the dimensionality increases (indeed the volume occupied by the points increases exponentially), thus, in the mentioned case, *nn-HilOut* takes a great advantage by storing the nearest points met during its execution.

Hence, from these experiments, when we deal with synthetic data sets, the *nn-HilOut* version appears to be superior to the *no-HilOut*.

Figures 12 (b) and 13 (b) report, in logarithmic scale, the execution times obtained varying the number n of top outliers to find from $n = 1$ to $N = 1,000$, for various values of d, for $N = 100,000$, and for $k = 100$. Figures 12 (e) and 13 (e) report, in logarithmic scale, the execution times obtained varying the number k of nearest neighbors from $k = 10$ to $k = 1,000$, for various values of d, for $N = 100,000$, and for $n = 100$.

Finally, we studied how the number of candidate outliers decreases during the algorithm.

Figure 12 (c) and 13 (c) report the number of candidate outliers at the beginning of each iteration of *nn-HilOut* for various values of the dimensionality d, for $N = 500,000$, and for $n, k = 100$. We note that, in the considered cases, if we fix the size of the data set and increase its dimensionality, then the ratio $\overline{d}/(d+1)$, where $\overline{d}$ is the number of iterations needed by the algorithm to find the solution, sensibly decreases, thus showing the very good behavior of the method for high dimensional data sets.

Figure 12 (f) and 13 (f) report the number of candidate outliers at the beginning of each iteration of *nn-HilOut* for various values of the data set size N, for $d = 128$, and for $n, k = 100$. These curves show that, at each iteration, the algorithm is able to discharge from the set of the candidate outliers a considerable fraction of the whole data set. Moreover, the same curves show that the algorithm terminates, in all the cases considered, performing much more less than the 129 iterations required, after 10 iterations for the Gaussian data set and only 4 for the Cluster data set.

6 Conclusions

We presented a new definition of distance-based outlier and an algorithm, called *HilOut*, designed to efficiently detect the top n outliers of a large and high-dimensional data set. The algorithm consists of two phases. The first phase provides an approximate solution with temporal cost $\mathcal{O}(d^2Nk)$ and spatial cost $\mathcal{O}(Nd)$. The second phase calculates the exact solution with a final scan. We presented both an in-memory and disk-based implementation of the *HilOut* algorithm to deal with data sets that cannot fit into main memory. Experimental results on real and synthetic data sets up to $500,000$ points in the 128-dimensional space showed that the algorithm always stops, reporting the exact solution, during the first phase, and that it scales well with respect to both the dimensionality and the size of the data set.

References

[1] C. C. Aggarwal and P.S. Yu. Outlier detection for high dimensional data. In *Proc. ACM Int. Conference on Managment of Data (SIGMOD'01)*, 2001.

[2] S. Aluru and F. E. Sevilgen. Parallel domain decomposition and load balancing using space-filling curves. In *Proceedings of the Int. Conf. on High Performace Computing*, pages 230–235, 1997.

[3] A. Arning, C. Aggarwal, and P. Raghavan. A linear method for deviation detection in large databases. In *Proc. Int. Conf. on Knowledge Discovery and Data Mining (KDD'96)*, pages 164–169, 1996.

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

[5] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *Proceedings of the Internatinal Conference on Database Theory*, pages 217–235, 1999.

[6] M. M. Breunig, H. Kriegel, R.T. Ng, and J. Sander. Lof: Identifying density-based local outliers. In *Proc. ACM Int. Conf. on Managment of Data (SIGMOD'00)*, 2000.

[7] C. E. Brodley and M. Friedl. Identifying and eliminating mislabeled training instances. In *Proc. National American Conf. on Artificial Intelligence (AAAI/IAAI 96)*, pages 799–805, 1996.

[8] T. Chan. Approximate nearest neighbor queries revisited. In *Proc. 13th Annual ACM Symp. on Computational Geometry*, pages 352–358, 1997.

[9] Yu D., Sheikholeslami S., and A. Zhang. Findout: Finding outliers in very large datasets. In *Tech. Report, 99-03, Univ. of New York, Buffalo*, pages 1–19, 1999.

[10] C. Faloutsos. Multiattribute hashing using gray codes. In *Proceedings ACM Int. Conference on Managment of Data (SIGMOD'86)*, pages 227–238, 1986.

[11] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proc. ACM Int. Conf. on Principles of Database Systems (PODS'89)*, pages 247–252, 1989.

[12] J. Han and M. Kamber. *Data Mining, Concepts and Technique.* Morgan Kaufmann, San Francisco, 2001.

[13] H.V. Jagadish. Linear clustering of objects with multiple atributes. In *Proc. ACM Int. Conf. on Managment of Data (SIGMOD'90)*, pages 332–342, 1990.

[14] H.V. Jagadish. Linear clustering of objects with multiple atributes. In *Proc. ACM Int. Conf. on Managment of Data (SIGMOD'90)*, pages 332–342, 1990.

[15] E. Knorr and R. Ng. Algorithms for mining distance-based outliers in large datasets. In *Proc. Int. Conf. on Very Large Databases (VLDB98)*, pages 392–403, 1998.

[16] E. Knorr, R. Ng, and V. Tucakov. Distance-based outlier: algorithms and applications. *VLDB Journal*, 8(3-4):237–253, 2000.

[17] D. E. Knuth. *The Art of Computer Programming, Vol.3 — Sorting and Searching.* Addison-Wesley (Reading MA), 1973.

[18] W. Lee, S.J. Stolfo, and K.W. Mok. Mining audit data to build intrusion detection models. In *Proc. Int. Conf on Knowledge Discovery and Data Mining (KDD-98)*, pages 66–72, 1998.

[19] M. Lopez and S. Liao. Finding k-closest-pairs efficiently for high dimensional data. In *Proc. 12th Canadian Conf. on Computational Geometry (CCCG)*, pages 197–204, 2000.

[20] B. Moon, H.V. Jagadish, C. Faloutsos, and J.H.Saltz. Analysis of the clustering properties of hilbert space-filling curve. *IEEE Trans. on Knowledge and Data Engineering (IEEE-TKDE)*, 13(1):124–141, Jan./Feb. 2001.

[21] F. Preparata and I. Shamos. *Computational geometry —An introduction—*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 2nd edition, 1985.

[22] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient algorithms for mining outliers from large data sets. In *Proc. ACM Int. Conf. on Managment of Data (SIGMOD'00)*, pages 427–438, 2000.

[23] S. Rosset, U. Murad, E. Neumann, Y. Idan, and G. Pinkas. Discovery of fraud rules for telecommunications-challenges and solutions. In *Proc. Int. Conf on Knowledge Discovery and Data Mining (KDD-99)*, pages 409–413, 1999.

[24] Hans Sagan. *Space Filling Curves.* Springer-Verlag, 1994.

[25] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-driven exploration of olap data cubes. In *Proc. Sixth Int. Conf on Extending Database Thecnology (EDBT)*, Valencia, Spain, March 1998.

[26] J. Shepherd, X. Zhu, and N. Megiddo. A fast indexing method for multidimensional nearest neighbor search. In *Proc. SPIE Conf. on Storage and Retrieval for image and video databases VII*, pages 350–355, 1999.

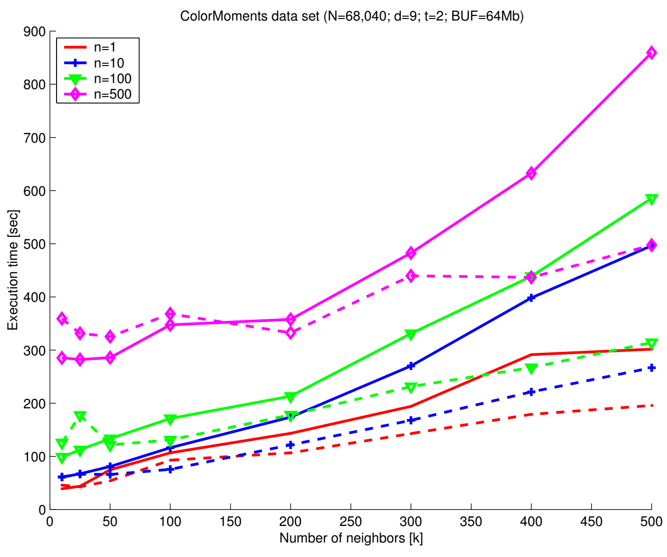
[27] Roman G. Strongin and Yaroslav D. Sergeyev. *Global Optimization with Non-Convex Costraints*. Kluwer Academic, 2000.

[28] Z.R. Struzik and A. Siebes. Outliers detection and localisation with wavelet based multifractal formalism. In *Tech. Report, CWI,Amsterdam, INS-R0008*, 2000.

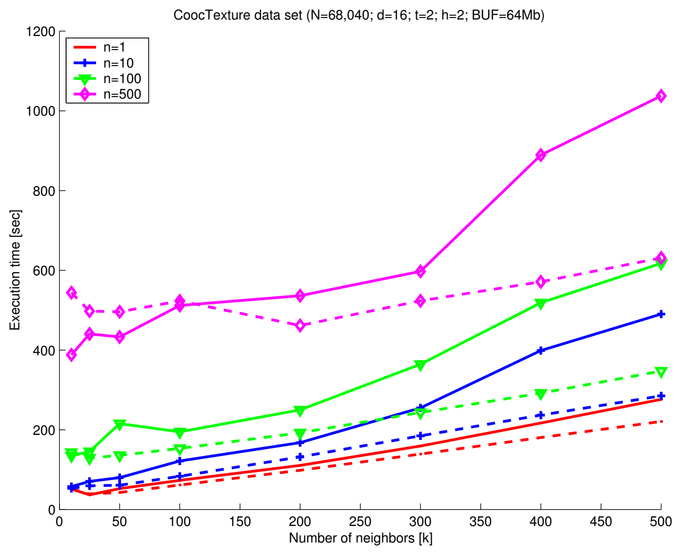
[29] K. Yamanishi and J. Takeuchi. Discovering outlier filtering rules from unlabeled data. In *Proc. ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 389–394, 2001.

[30] K. Yamanishi, J. Takeuchi, G.Williams, and P. Milne. On-line unsupervised learning outlier detection using finite mixtures with discounting learning algorithms. In *Proc. ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 250–254, 2000.

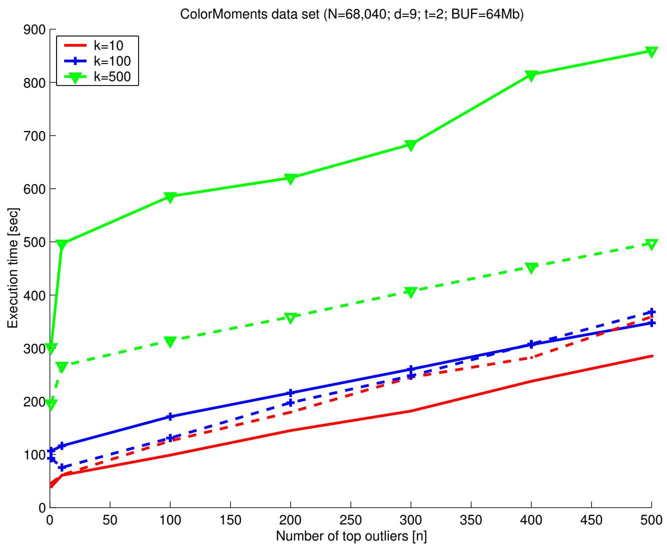
[31] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. In *Proceedings of the ACM SIGMOD Int. Conf. on Managment of Data*, pages 103–114, 1996.



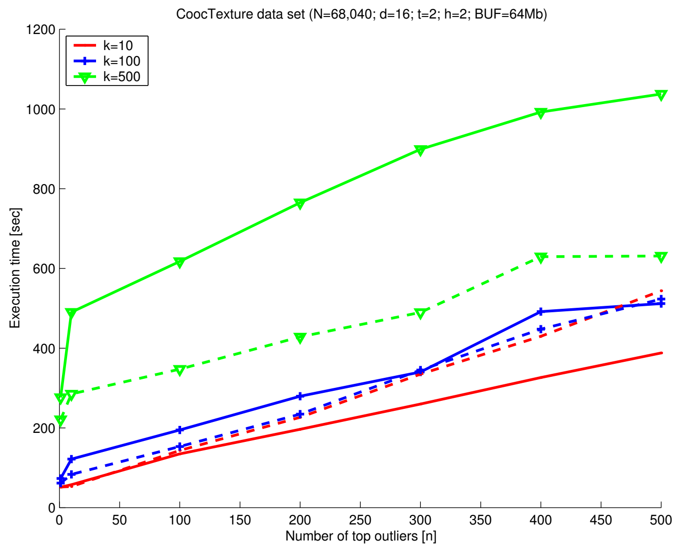
(a)



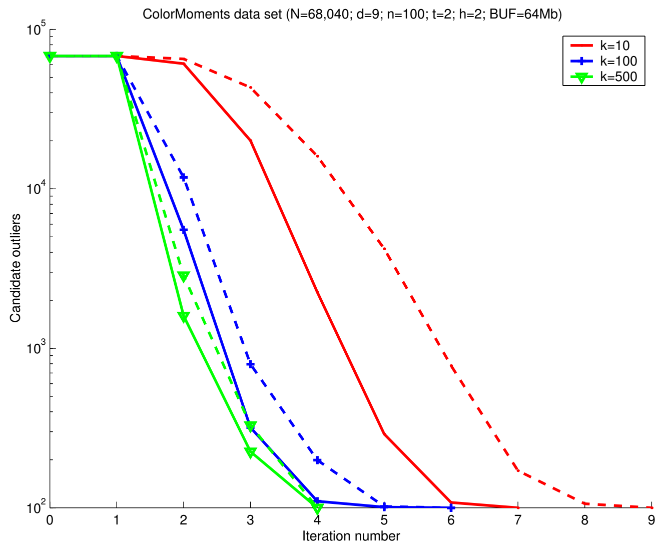
(d)



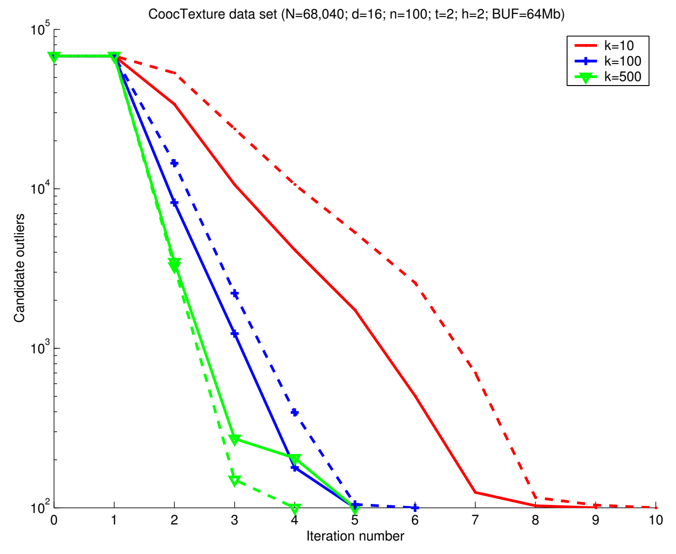
(b)



(e)

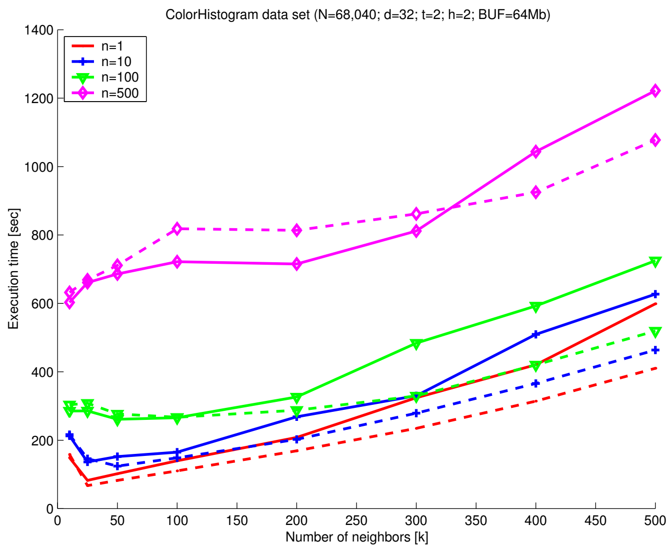


(c)

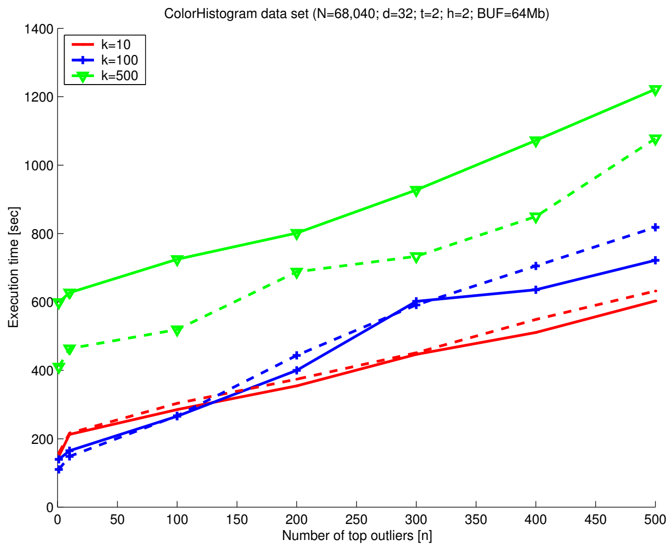


(f)

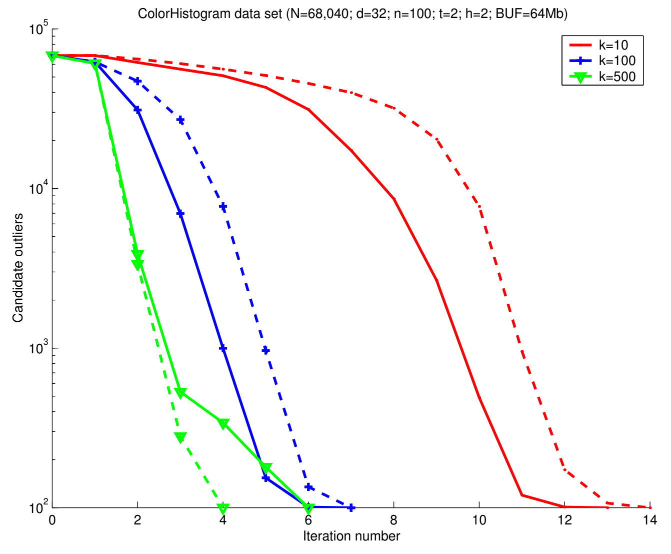
Figure 9: Experimental results - Part 1



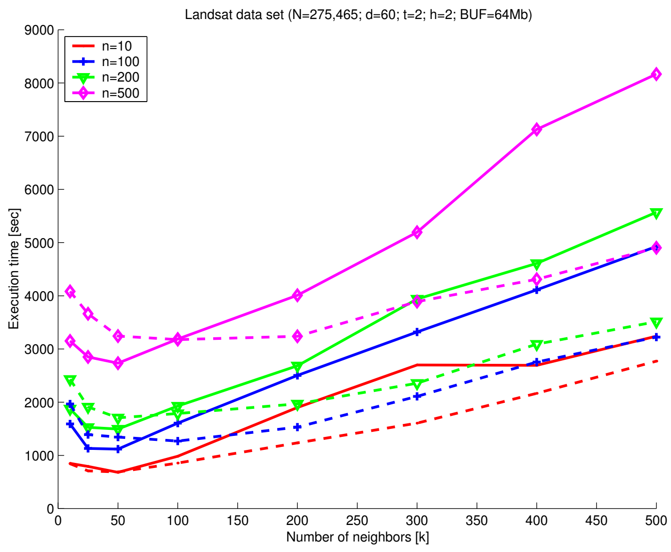
(a)



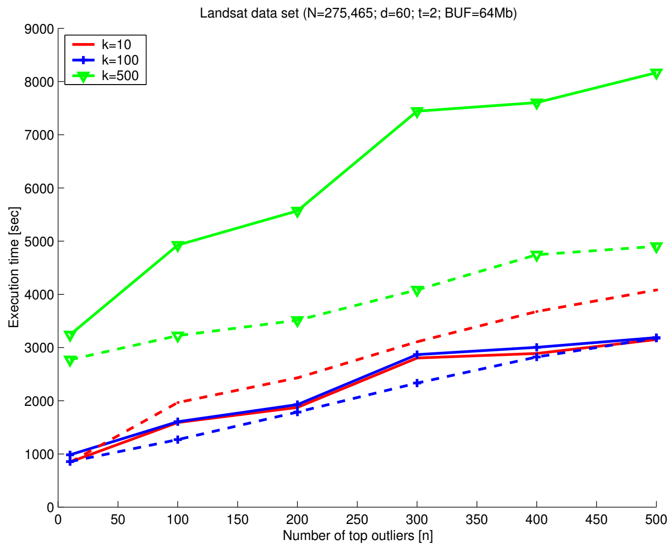
(b)



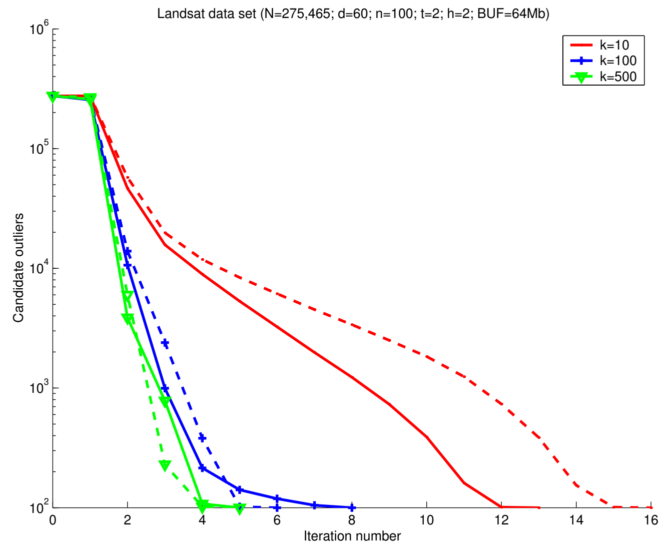
(c)



(d)

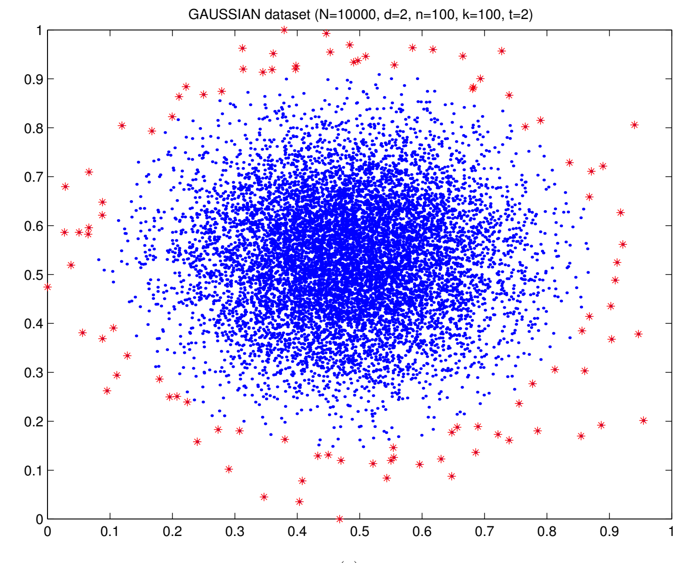


(e)

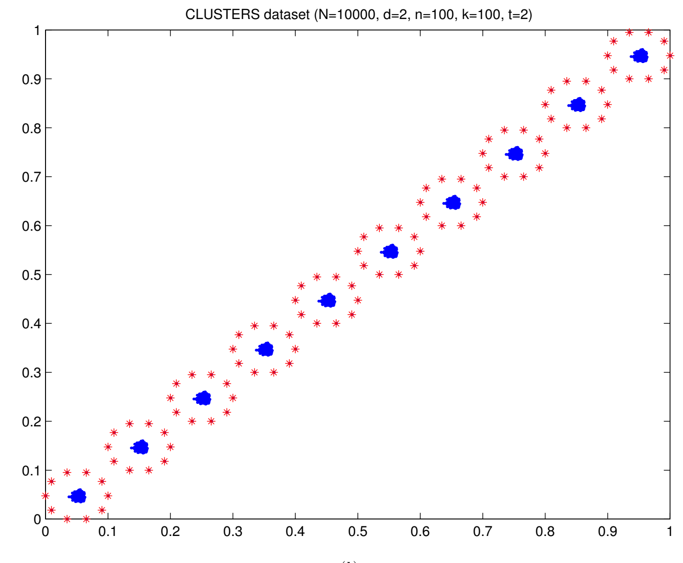


(f)

Figure 10: Experimental results - Part 2

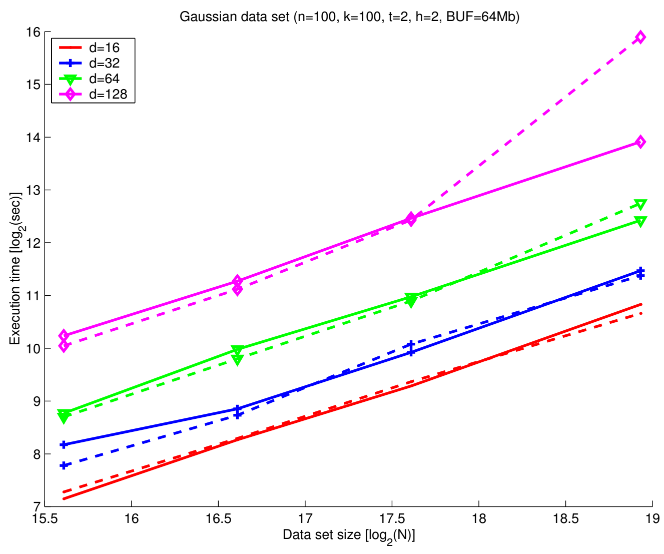


(a)

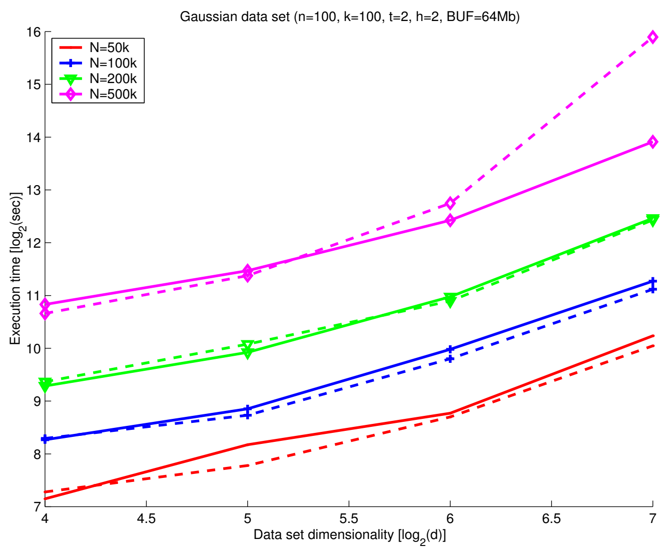


(b)

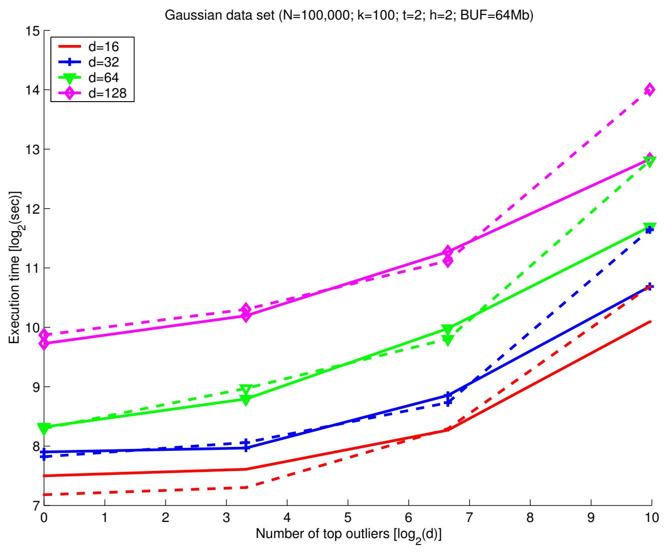
Figure 11: Synthetic data sets



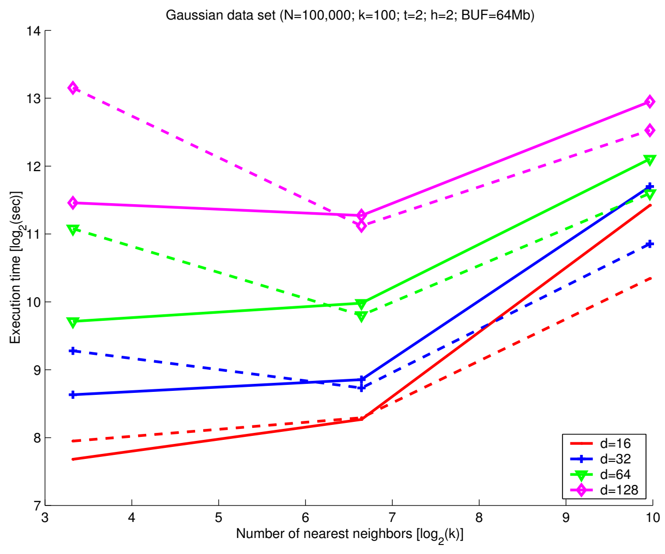
(a)



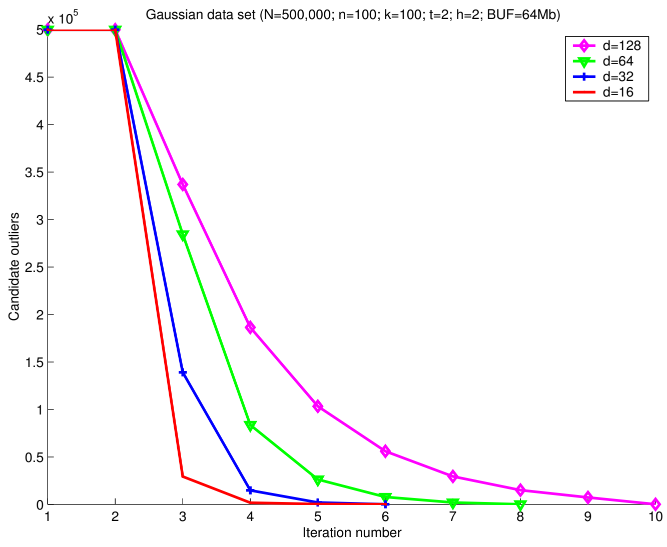
(d)



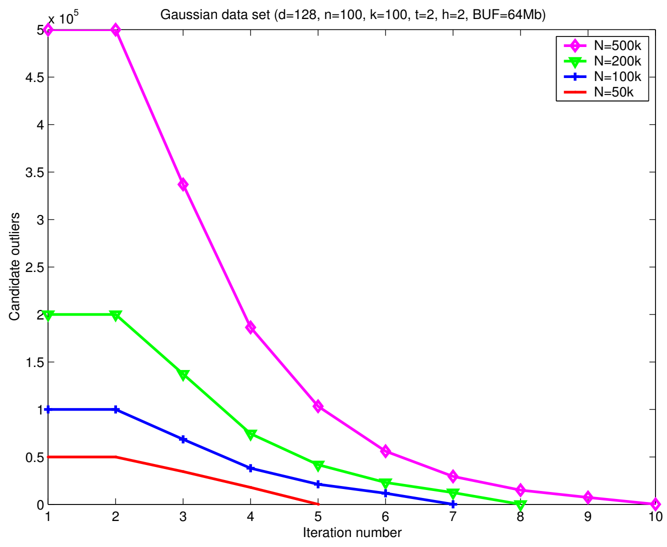
(b)



(e)

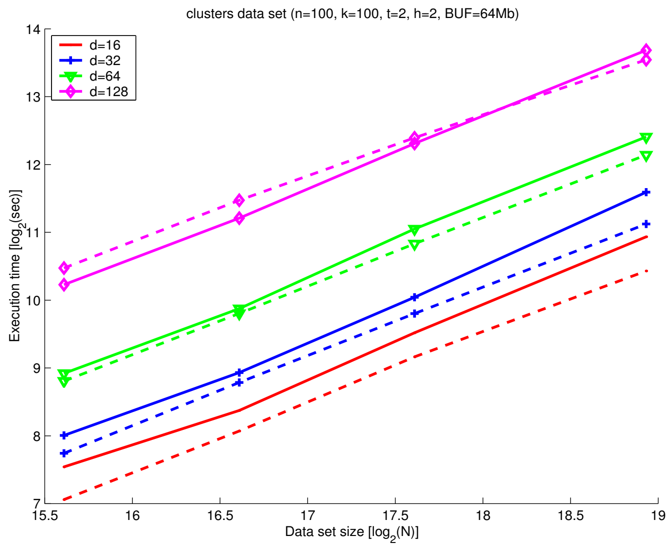


(c)

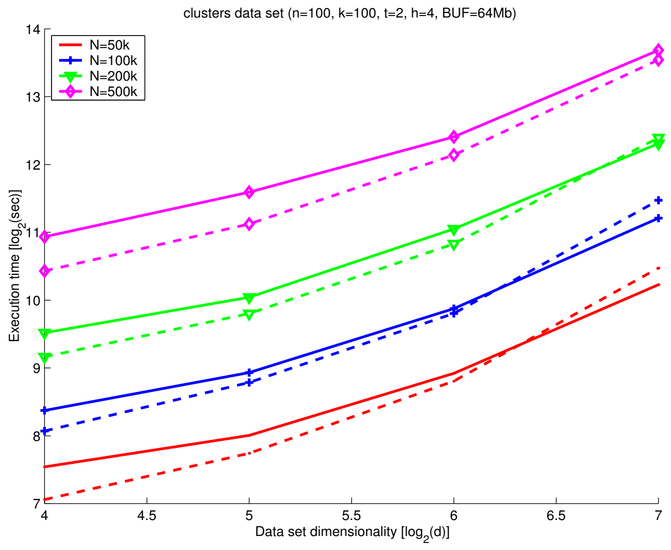


(f)

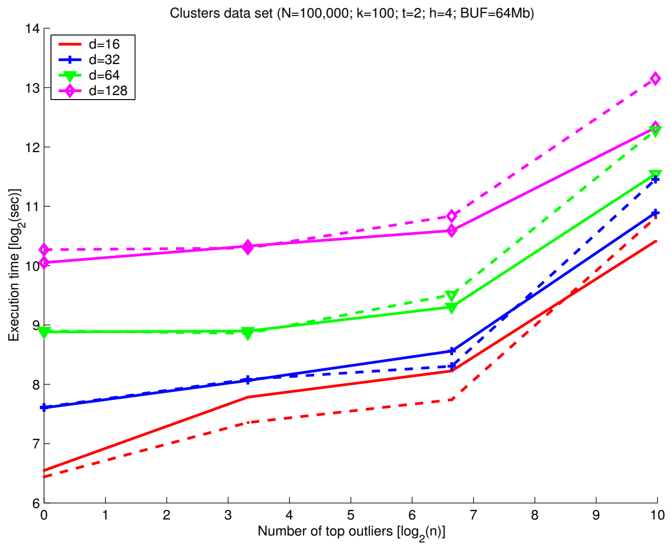
Figure 12: Experimental results: *Gaussian* data set



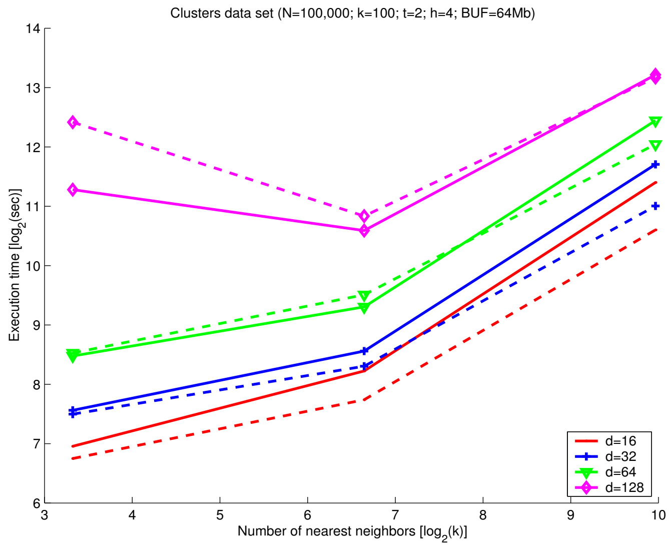
(*a*)



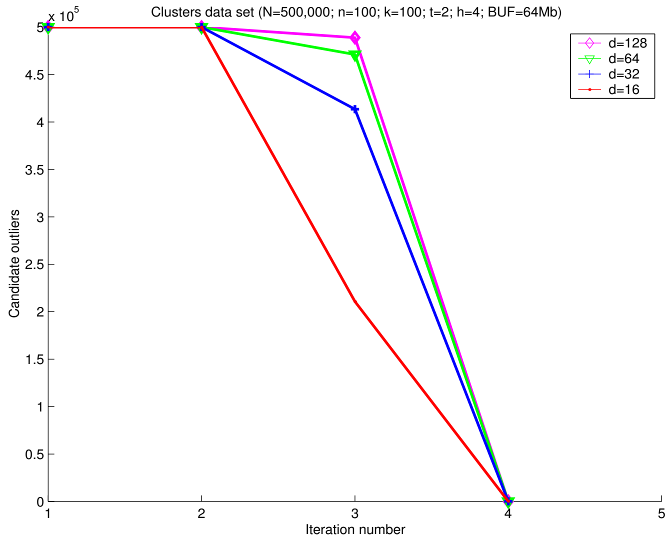
(*d*)



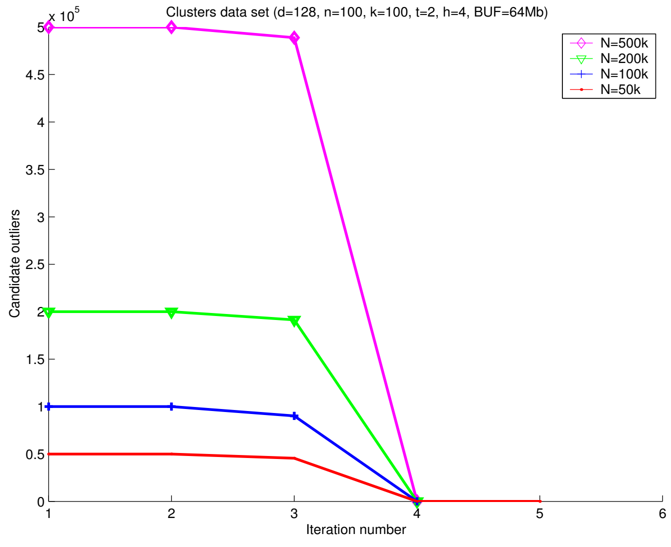
(*b*)



(*e*)



(*c*)



(*f*)

Figure 13: Experimental results: *Clusters* data set