Open access • Proceedings Article • DOI:10.1145/2020408.2020554

Algorithms for speeding up distance-based outlier detection — Source link

Kanishka Bhaduri, Bryan Matthews, Chris Giannella

Institutions: Ames Research Center, Mitre Corporation

Published on: 21 Aug 2011 - Knowledge Discovery and Data Mining

Topics: Distributed algorithm, Sequential algorithm, Outlier, Time complexity and Sorting

Algorithms for Speeding up Distance-Based Outlier Detection

Kanishka Bhaduri
MCT Inc., NASA Ames Research Center
Moffett Field, CA 94035 USA
Kanishka.Bhaduri-1@nasa.gov

Bryan L. Matthews
SGT Inc., NASA Ames Research Center
Moffett Field, CA 94035 USA
Bryan.L.Matthews@nasa.gov

Chris R. Giannella
The MITRE Corp.
300 Sentinel Dr., Suite 600
Annapolis Junction, MD 20701 USA
cgiannel@acm.org

ABSTRACT

The problem of distance-based outlier detection is difficult to solve efficiently in very large datasets because of potential quadratic time complexity. We address this problem and develop sequential and distributed algorithms that are significantly more efficient than state-of-the-art methods while still guaranteeing the same outliers. By combining simple but effective indexing and disk block accessing techniques, we have developed a sequential algorithm *iOrca* that is up to an order-of-magnitude faster than the state-of-the-art. The indexing scheme is based on sorting the data points in order of increasing distance from a fixed reference point and then accessing those points based on this sorted order. To speed up the basic outlier detection technique, we develop two distributed algorithms (*DOoR* and *iDOoR*) for modern distributed multi-core clusters of machines, connected on a ring topology. The first algorithm passes data blocks from each machine around the ring, incrementally updating the nearest neighbors of the points passed. By maintaining a cutoff threshold, it is able to prune a large number of points in a distributed fashion. The second distributed algorithm extends this basic idea with the indexing scheme discussed earlier. In our experiments, both distributed algorithms exhibit significant improvements compared to the state-of-the-art distributed method [13].

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database applications—*Data Mining*

General Terms

Algorithms, Experimentation, Performance

Keywords

Outlier detection, Distributed computing, Nearest neighbor

KDD'11, August 21–24, 2011, San Diego, California, USA.

1. INTRODUCTION

Unsupervised outlier detection is an important and well-studied problem in the data mining, machine learning, and statistics literature. Given the broad array of meanings that may be sensibly ascribed to the term "outlier", many variations of the problem have been developed and addressed. One commonly used definition in the literature involves defining points as outliers if their nearest neighbors are sufficiently distant (distance-based outliers). A specific form of the problem along these lines can be stated by first defining an *outlier score* for each point x in dataset D as the distance to the kth nearest neighbor of x in D [3] or, similarly, the sum or average of the distances of the k nearest neighbors of x in D [15]. Then the outlier detection problem is as follows: given parameters t and k, find the top t points in D in terms of their outlier scores[1]. This definition has been used in many domains to find anomalies such as Earth sciences, astronomy, space applications and more [8].

The naive nested loop algorithm requires $O(n^2)$ (with $n = |D|$) computation time which is infeasible for large datasets. A key idea for improvement, developed by Bay and Schwabacher [4] (the *Orca* algorithm), is to maintain the t-th largest outlier score and use it as a *cutoff threshold*. When considering a point x, if the distance to the k-th nearest neighbor found thus far is less than the threshold, then x can be immediately pruned. A key to making this idea work even better is considering the points in D in an order such that the outliers are considered as early as possible. Doing so allows the cutoff threshold to be set as high as possible, as early as possible in the algorithm. One of the key contributions of our work is a simple and quick-to-build index that allows the points to be considered in an order that significantly improves upon the random ordering of Orca [4]. The index is a list of two-tuples consisting of IDs of data points ordered by decreasing distance to a randomly chosen reference point and the respective distances. For a given test point x, we search on the index for finding its k nearest neighbors by gradually "spiraling out" along the index. Based on a new termination criterion that we develop, the

[1]All of our algorithms can be used with either of these two definitions of outlier score. For simplicity, henceforth we will work with the k nearest neighbor definition. A closely related version of the problem we study was developed in [12]: given parameters r and k, find all points in D whose outlier score is less than r. However, [15] argue that r can be difficult to set and is dataset dependent. We do not consider the "r-version" of the problem further in this paper.

algorithm can terminate without having to scan the entire dataset. These contributions combine to form *iOrca*, a new outlier detection algorithm.

We also develop two distributed algorithms both on top of a ring overlay network leveraging the modern multi-core cluster of machines. The first algorithm passes data blocks around the ring allowing the computation of k nearest neighbors to proceed in a parallel. Along the way, the cutoff threshold is updated and distributed across all nodes. As such, data points can be pruned at any node without central coordination. This idea allows the cutoff threshold to be utilized fully and efficiently and offers a significant improvement over the distributed parallel Bay's algorithm (PBay) [13]. The second distributed algorithm extends this basic idea with the indexing scheme discussed earlier.

Our main contributions: (1) We develop a new outlier detection algorithm, *iOrca*, which improves upon the Orca algorithm [4] by means of a novel indexing scheme. The index allows for greater pruning of data points based on the cutoff threshold and also allows for more efficient computation of the k nearest neighbors of any given datapoint. Furthermore, it allows us to terminate the algorithm much earlier, without even looking at all the data points for outliers, thereby saving both I/O cost and runtime. (2) We develop two new distributed outlier detection algorithms based on a ring computation which allows the cutoff threshold to be exploited efficiently in a distributed fashion. To the best of the authors' knowledge, this paper shows for the first time how indexing can be used in conjunction with distributed processing to speed up distance-based outlier detection technique. (3) We experimentally demonstrate that our sequential and distributed algorithms are significantly more efficient than the respective state-of-the-art methods.

2. RELATED WORK

The naive approach of distance-based outlier detection takes quadratic time with respect to the number of points in the dataset by comparing each point with the rest. Such a technique has been proposed by Knorr and Ng [12]. Their first algorithm is a block nested loop algorithm which scales as $O(n^2d)$, where n is the size and d is the dimensionality of the dataset. Their second algorithm partitions the dataset into cells that are hyper-rectangles. By efficiently pruning the unpromising rectangles early, their second algorithm executes in time linear in n but exponential in d. Hence, both these algorithms do not work well for large datasets.

Spatial database indexing methods such as KD-tree [5], R-tree [10] or X-tree [6] can be used for speeding up nearest neighbor search. The main idea is to query the k-nearest neighbors of each point in the index structure. This method works extremely well for low dimensional datasets, often scaling as $O(n \log n)$. However, they scale poorly with the dimension of the data, making them inefficient for dimension greater than 10 [7].

Ordering the data often helps in the nearest neighbor search for outlier detection. Knorr and Ng [12] present an idea in which the data is first split into multiple cells (hyper rectangles) and only the corners of the minimum bounding rectangles (MBRs) are stored. For any test point, all those MBRs are pruned which cannot contain the nearest neighbors of the test point. One major drawback of this approach is that the complexity of these hyper rectangles increases exponentially with the dimension of the dataset, rendering them ineffective for high dimensional datasets. Similar problems with R-tree based indexing schemes have also been reported by Ramaswamy *et al.* [15].

Orca proposed by Bay and Schwabacher [4] shows how distance-based outlier detection can be made efficient using a simple pruning strategy and randomization of the dataset. We describe it in more detail in Section 3.3 since this forms a baseline for our comparison. To speed up Orca, Ghoting *et al.* [9] have proposed the Recursive Binning and Re-Projection (RBRP) method. In their method, the data is first clustered and then, for any test point x_t, the nearest neighbors of x_t are first searched in the cluster in which x_t belongs to and then in the next closest cluster and so on. The authors empirically show that this method speeds up Orca. However, this method needs to load the entire dataset in memory during the clustering phase which can be a serious drawback for large and high dimensional datasets. Also, for each test point, RBRP needs to go through the data in a particular order, leading to an excessive number of disk I/O. Furthermore, RBRP does not reorder the test data points and so the growth in the cutoff rate is much slower compared to *iOrca*. Another technique for improving Orca is the Dolphin algorithm proposed by Angiulli and Fassetti [2]. In their method, using a pivot-based index, they only need two sequential scans of the dataset, leading to an improved running time with respect to Orca. However, the authors use the "r" definition of outliers and hence do not find the same outliers as Orca or *iOrca*. Moreover, [2] does not demonstrate the accuracy of their method compared to Orca or RBRP.

There exist some approaches to speeding up distance-based outlier detection methods using parallel/distributed computing. In the PBay algorithm by Lozano and Acuna [13], a master node first splits the data into separate chunks for each processor. Then the master node loads each block of test data and broadcasts it to each of the worker nodes. Each worker node then executes Orca using its local database and the test block. The nearest neighbors of the test points from all the workers are aggregated at the master node to find the global set of nearest neighbors for those test points. All points whose score is less than the cutoff are dropped and the cutoff is updated accordingly. The cutoff is broadcast back to all the worker nodes along with the next block of test data. The algorithm that we propose in this paper distributes workload among the nodes more efficiently and does not require the master node to do all the aggregation of the nearest neighbors; as a result both of our distributed algorithms are much faster. Hung and Cheung [11] present a parallel version of the basic nested loop algorithm which is not suitable for distributed computation since it requires all the dataset to be exchanged among all the nodes. Otey *et al.* [14] present a distributed algorithm in which the definition of outlier is completely different than the one presented here. In order to process mixed attributes, their definition is based on support. In their work, outlier score function is defined by taking into account the dependencies among all the attributes in the dataset and outliers are those points which contradict these dependencies. Angiulli *et al.* [1] present a distributed distance-based outlier detection algorithm based on the concept of *solving set* which can be viewed as a compact representation of the original dataset. The solving set is such that by comparing any data point to only the elements of the solving set, it can be concluded if the point

is an outlier or not. This solving set is itself updated as more and more points are processed such that it at least contains the correct scores of the top t requested outliers. They demonstrated the performance of their algorithm on only two real datasets which shows linear scalability (wrt number of nodes). Our algorithm combines both distributed processing and indexing, thereby giving us better speedup.

3. BACKGROUND

3.1 Notations

Let $t, k > 0$ be fixed (user-defined) parameters. Let D be a finite subset of $\mathbb{R}^d$ having n points such that $|D| \geq \max\{t, k\}$. Given $x \in D$, let $N_k(x, D)$ denote the set of k nearest neighbors from $\{D \setminus \{x\}\}$ to x (with respect to Euclidean distance with ties broken according to some arbitrary but fixed total ordering $\prec$).

Let $\delta_k(x, D)$ denote the maximum distance between x and all the points in $N_k(x, D)$ *i.e.* the distance between x and its k-nearest neighbors in D. $\delta_k(x, D)$ can be viewed as an *outlier ranking function*. Let $O_{t,k}(D)$ denote the top t points (outliers) in D according to $\delta_k(., D)$. In the rest of the description, for simplicity, we rewrite $N_k(b, D)$, $\delta_k(b, D)$ and $O_{t,k}(D)$ as $N_k(b)$, $\delta_k(b)$ and O_k.

Definition 3.1 (Centralized problem definition). *Given integers $t, k > 0$, and dataset D, the goal of outlier detection algorithm is to compute the outliers O_k in D as quickly as possible.*

3.2 Distributed problem definition

In the distributed setup, there are p machines $P_1, \ldots, P_p$ connected to each other in a ring $\mathcal{G}$, such that P_i's neighbor Γ_i is known to P_i. Each P_i holds a dataset D_i such that size of $|D_i| \geq \{t, k\}$. We assume that the datasets are disjoint *i.e.* $D_i \cap D_j = \emptyset$, $\forall i \neq j$.

Definition 3.2 (Distributed problem definition). *Given integers $t, k > 0$, and dataset D_i at each node P_i, the goal of outlier detection algorithm is to compute the outliers O_k (in D) as quickly as possible, where $D = \bigcup_{i=1}^{p} D_i$.*

In the above definition, we have assumed that the distributed outlier detection algorithm produces the same set of outliers as Orca [4]. The algorithms that we have proposed in this paper all guarantee that the set of outliers are the same as Orca, therefore, we drop the issue of accuracy from our problem definition.

3.3 Distance-based outlier detection with pruning: Orca

In this section we describe Orca [4] in detail since it forms our baseline for comparison. Orca achieves a near-linear runtime for many datasets by using a simple pruning rule and randomization. It is disk-based and processes the data in blocks instead of reading in all the data at once from memory. The pseudo-code of the algorithm is shown in Alg. 1. The inputs are k, t, the block size b, and the dataset D. The basic idea of the algorithm is very simple. Let cutoff c denote the score of the t-th largest outlier (or 0 if less than t points have been processed). It reads in a block of data from the memory (call it the test block) and then computes the nearest neighbors of each point in the test block by looking at all the other points. During this computation it keeps track of the nearest neighbors found so far for each test point. As soon as the outlier score (*e.g.* the distance to the k-th nearest neighbor) drops below c, that point is pruned since it can no longer be an outlier. If any outlier is found for this block, both the outlier list and the cutoff are updated. As more points are processed, c increases, and the algorithm achieves a better prune rate.

In Orca, the number of disk access is $O(n/b * n)$ where $n = |D|$. Also, as shown in [4], the expected number of distance computations to prune a non-outlier point x is $1/\rho_x$, where ρ_x is the probability that a point chosen at random lies within the cutoff threshold. Since this is independent of n, it achieves a near linear time performance on the entire dataset. However, to achieve this, ρ_x needs to be a constant. This does not hold for many datasets as we demonstrate later. Our indexing technique described in the next section helps to alleviate this problem by ensuring that the cutoff grows faster, the nearest neighbors of a test point are found quicker and the algorithm terminates earlier without having to scan the entire database.

```
Input: k, t, b, D
Output: O_k, the set of top t outliers in D
Initialization: c ← 0, O ← ∅
while B ←get_next_block(D, b)<> ∅ do
    forall the b ∈ B do N_k(b) ← ∅;
    forall the i = 1 to |D| do
        x = getFromfile(i, D);
        forall the b ∈ B, b ≠ x do
            if |N_k(b)| < k or dist(b, x) <
            maxdist(b, N_k(b)) then
                N_k(b) ← Update_nbors(N_k(b), x, k);
                δ_k(b) ← max{‖b − y‖ : y ∈ N_k(b)};
                if δ_k(b) < c then B ← B \ b;
            end
        end
    end
    for b=1 to B do
        newO ← newO ⋃[b;  δ_k(b);  N_k(b)];
    end
    O_k ← Find_Top_t(newO ∪ O_k, t);
    c ← min{δ_k(y, D) : y ∈ O};
end
```

Algorithm 1: Orca by Bay and Schwabacher [4].

4. FAST OUTLIER DETECTION USING INDEXING: IORCA

In Orca, one of the factors that affects the prune rate is the cutoff threshold. If the cutoff grows very slowly, it does many redundant comparisons for inliers. To alleviate this problem, we need to develop a method such that: (1) the cutoff is updated faster, (2) the data is rearranged such that for every point, the nearest neighbors are found quickly, (3) it does not perform unnecessary disk accesses in order to achieve (1) and (2), and (4) the index is fast to build and does not require the entire data to be loaded in memory. In the remainder of this section we show how our indexing method addresses each of these problems elegantly.

In order to update the cutoff faster, we need to process the outliers as early as possible. In our indexing method, we first select a random point R from D as a reference point

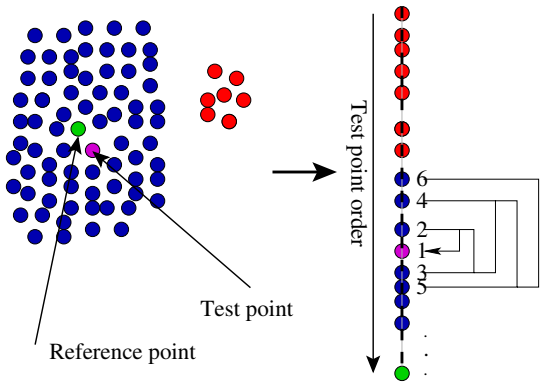


Figure 1: Description of the index. Left figure shows a dataset with normal points in blue, outliers in red and the reference and test point. The right figure shows the order in which the test points are processed with the points farthest from the reference point being processed first.

and then calculate the distance of all the other points in D from R. Then the points in the database are reordered with increasing distance from R. Fig. 1 shows the index for a dataset. The left part shows the original dataset. The blue circles represent the inlier points while the red ones denote the outliers. The green circle at the center is the randomly chosen reference point. The right part of the figure shows the index. The index is simply a one dimensional list of ordered points where the ordering is determined by the distance of all the points in the database from the reference point (highest to lowest). Instead of traversing through the data in the original order (or random order [4]), we test the points along this index order. The rationale for this traversal is as follows. Since we choose R at random, it is likely that R will be one of the inlier points if there are more inliers than outliers. Therefore, the points farthest from R will be more likely be the outliers. Since we process the points in decreasing distance to R, it is very likely that the outliers will be processed first, leading to a faster increase in the cutoff threshold. This approach would therefore, satisfy challenge (1) as stated above.

Re-ordering the database for testing has another advantage. If the points are processed in this order, it can be shown that *iOrca* can stop the processing and terminate much early compared to other state of the art methods by using the following termination criterion.

LEMMA 4.1. *[Stopping rule] Let L be the index as described in this section. Let R be the reference point used to build L. Let x_t be any test point currently being tested by iOrca. If*

$$\|x_t - R\| + \|R - x_{n-k+1}\| < c,$$

then iOrca can terminate the execution immediately, where c is the current cutoff threshold, x_{n-k+1} is the true k-th nearest neighbor of R and $\|\cdot\|$ is the 2-norm of a vector.

PROOF. By applying triangle inequality for every point x_j, where $n-k+1 \leq j \leq n$, it follows that,

$$\|x_t - x_j\| < \|x_t - R\| + \|R - x_j\| < c$$

i.e. there exist at least k points between x_{n-k+1} and x_n whose distance to x_t is less than c. This implies that the distance between x_t and its k-th nearest neighbor is less than c, in which case it can be pruned. Surprisingly, the entire execution of the algorithm can be terminated at this point since, for any other test point x'_t closer to y than x^t (and hence not yet tested), we can show that,

$$\begin{aligned} \left\|x'_t - x_{n-k+1}\right\| &< \left\|x'_t - y\right\| + \|x_{n-k+1} - y\| \\ &< \|x_t - y\| + \|x_{n-k+1} - y\| < c \end{aligned}$$

This shows that if the result holds for any test point, it will hold for all subsequent test points and so the algorithm can stop without further testing of any other point. □

Note that these distances are pre-computed and available in memory. Therefore, checking if the condition holds is extremely fast and causes very little overhead.

In order to quickly find the nearest neighbors of a test point, we again apply our index. Instead of starting the search from the beginning of the database for every test point, we start the search from the location it lies along the index and then gradually "spiraling" out. Figure 1 shows one of the test points T (magenta color) in the dataset. It also shows where T lies along the index. Now the search for the nearest neighbor spirals out from T by first looking at the nearest two points (top and bottom) and then continuing further until the score of that point drops below the threshold (in which case the point is pruned) or the test point has a score higher than the t-th largest outlier (in which case the outlier list and the cutoff are updated). Since the index prescribes a total ordering of the points projected along one dimension, it is likely that, for a test point x_t, if x_i is closer to x_t than x_j along the index, then it will be so when the actual distances are also computed *i.e.* if

$$\|x_t - y\| > \|x_i - y\| > \|x_j - y\|$$

then it is expected that

$$\|x_t - x_i\| < \|x_t - x_j\|.$$

As a result, we can find the nearest neighbors very fast as demonstrated by our extensive experimental study.

The above technique prescribes a way of testing each point by traversing the database points in a spiral order. Loading the entire database for testing each point one by one is I/O expensive; instead, we process the test points in blocks such that they can be tested simultaneously by creating a spiral search order for that entire test block. Our index allows us to do this by loading a block of index distances (they are the distances of the test points to y), computing the average distance and then starting the spiral search from the location where this average lies along the index. Since after ordering, the test blocks are expected to be similar, testing them using the same spiral order helps speed up the search as well.

Finally, our index is simple and cheap to compute. It can be entirely loaded in memory at runtime since it only consists of n floating point numbers and the associate integer indices.

Putting these building blocks together, we develop a novel indexed version of Orca (*iOrca*) which is upto an order of magnitude faster than Orca and costs less disk I/O. The pseudo-code of *iOrca* is shown in Alg. 2. Note that the choice of R is critical in determining the effectiveness of the pruning. However, discussion on a proper choice of R is beyond the scope of this paper.

```
Input: k, t, b, D
Output: O_k, the set of top t outliers in D
Initialization: c ← 0, O_k ← ∅, L = BuildIndex(D);
while B ←get_next_block(D, b, L)<> ∅ do
        // process points as specified in L.id
    if (Lemma 4.1 holds for B(1)) then Terminate;
    else
        μ = findAvg(L(B));
        startID =find(L >= μ);
        order = spiralOrder(L.id, startID);
        forall the b ∈ B do N_k(b) ← ∅;
        forall the i = 1 to |D| do
            x = getFromfile(order(i), D);
            forall the b ∈ B, b ≠ x do
                if |N_k(b)| < k or dist(b, x) <
                maxdist(b, N_k(b)) then
                    N_k(b) ← Update_nbors(N_k(b), x, k);
                    δ_k(b) ← max{||b − y|| : y ∈ N_k(b)};
                    if δ_k(b) < c then B ← B \ b;
                end
            end
        end
        for b=1 to B do
            newO ← newO ⋃ [b;  δ_k(b);  N_k(b)];
        end
        O_k ← Find_Top_t(newO ∪ O_k, t);
        c ← min{δ_k(y, D) : y ∈ O};
    end
end
```

Algorithm 2: Indexed Orca (*iOrca*)

Complexity analysis: For building the index, n distance computations are needed, each taking $O(d)$ time. Also, the sorting takes $O(n \log n)$ time, giving a total running time of $O(n \log n + nd)$. n floating point numbers are required to be stored as the index, so the space requirement is $O(n)$. Since we block both the test points and the reference points using a block size of b, the number of disk accesses is $O(n/b * n/b)$ in the worst case, unless the algorithm can terminate early.

5. DISTRIBUTED OUTLIER DETECTION ALGORITHM: DOOR

In this section we describe two distributed algorithms for distance-based outlier detection (*DOoR*). The algorithms are disk-aware, and substantially faster than the state of the art methods while still guaranteeing correct results.

In our distributed setup, we assume that there is a central node which does the reporting of all the outliers. We also assume that all nodes $P_1, \ldots, P_n$ in $\mathcal{G}$ form a ring (except the leader node P_0) *i.e.* any node P_i can communicate with nodes P_i+1 and P_i-1, $1 \leq i < p$. It is further assumed that the leader has access to all the data; splits the dataset into n partitions and ships them to the nodes, while still maintaining a copy for itself to be used as the test points. Each P_i has access to its own data block D_i; the test blocks are supplied either by the leader or read from the disk. P_0 maintains the current list of outliers O_k, of at most t points which currently have the largest $\delta_k(, D)$. Initially O_k is empty and accumulates more points as their k-nearest neighbors w.r.t D are computed by other machines. The leader also maintains a cutoff threshold, $c = \min\{\delta_k(x, D) : x \in O\}$, with $c = -\infty$ initially. Whenever this threshold is changed, it is broadcast to all nodes for more efficient pruning of points.

```
Input: k, t, b, D_i
while B ←get_next_block(D_i, b)≠ ∅ do
    forall the b ∈ B do L_k(b) ← ∅;
    forall the ℓ = 1 to |D_i| do
        x = getFromfile(ℓ, D_i);
        forall the b ∈ B, b ≠ x do
            if |L_k(b)| < k or dist(b, x) <
            maxdist(b, L_k(b)) then
                L_k(b) ← Update_nbors(L_k(b), x, k);
                r_b ← max{||b − y|| : y ∈ L_k(b)};
                if r_b < c_i then B ← B \ b; τ_i ← τ_i + 1;
            end
        end
    end
    forall the b ∈ B do
        Send (b, L_k(b), r_b) to machine (i + 1) mod p;
    end
    CheckMsg();
    CheckThreshold();
end
Procedure CheckMsg()
begin
    while (received buffer ≠ ∅) do
        Extract (x, L_k(x), r_x) from received buffer;
        L_k(x) ← N_k(x, D_i) ∪ L_k(x);
        r_x ← max{||x − y|| : y ∈ L_k(x)};
        if r_x > c_i then
            if x originated in machine P_i then
                Send (x, r_x, L_k(x)) (a potential outlier
                message) to the leader machine (P_0);
            end
            else Send (x, L_k(x), r_x) to machine (i + 1)
            mod p (put it in received buffer of machine
            (i + 1) mod p);
        end
        else τ_i ← τ_i + 1;
    end
end
On receiving a new threshold c: c_i ← c;
```

Algorithm 3: *DOoR* at any worker node P_i.

Each worker node works in a *push-pull* mode. Description of the worker node for *DOoR* is given in Fig. 2 and Alg. 3. The inputs to each worker are k, t, size of the block b, and the local dataset D_i. Machine P_i maintains a threshold c_i it has received from the leader. Initially $c_i = -\infty$. For each point b in the test data block B, machine P_i also maintains:

- $\mathcal{L}_k(b)$— the k-nearest neighbors found thus far for b
- $r_b = \max\{\|b - y\| : y \in \mathcal{L}_k(b)\}$

Initially, $\mathcal{L}_k(b) \leftarrow \emptyset$ and $r_b = 0$ for each point $b \in B$. In the *push* mode, for each point $b \in B$, the algorithm checks to see if there exist a set of neighbors in $\mathcal{L}_k(b)$ such that the current score of b is below c_i *i.e.* if $r_b < c_i$. If this is true, then the point is no longer tested and pruned; otherwise b along with its nearest neighbors found so far $\mathcal{L}_k(b)$ and r_b are forwarded (*pushed*) to the next node P_{i+1} for validation.

The other phase of *DOoR* is the *pull* phase in which P_i first checks to see if there are any messages in the received

buffer (RB) from node P_{i-1}. If RB is not empty, P_i needs to validate all the received points against its local data D_i. It does so by first extracting the points from RB and then checking, for every point x in RB, if there are any neighbors in D_i which are closer than the ones in $\mathcal{L}_x$ (which contains the current closest neighbors as received from P_{i-1}). The neighbor list and the value of r_x are updated accordingly. As a result, if r_x becomes less than c_i, then x is pruned. Otherwise, if x originated in P_i itself, it has survived the pruning of all the nodes and is sent to the leader node P_0 (since it can be a potential outlier data point). If x did not originate on P_i, is forwarded to P_{i+1} with the updated nearest neighbors. Node P_i then goes back to the push mode and begins testing the new set of points. In any step of the execution, if any node gets a new cutoff threshold c, it immediately sets $c_i \leftarrow c$ and resumes the processing.

Alg. 4 shows the tasks executed by the leader node in *DOoR*. It initializes the outlier list O_k to null. Whenever it receives a new potential outlier x, it adds x to O_k if the outlier list contains less than t points. If it contains t points, the outlier in O_k with the smallest score is replaced by x. If due to either of these computations, the outlier list becomes full, the cutoff is set to the score of the smallest outlier and it is then broadcast to all the nodes.

Finally, we discuss the assignment of test points to the worker nodes. There are several ways this can be done. In our implementation, we choose a simple but effective approach. This is shown in Fig. 2. Instead of testing all the test blocks in parallel at all nodes, we assign the test blocks in a round robin fashion: the first test block (red) is assigned to the top left node, the second to the top right and the third goes to the bottom node. These points are then tested in parallel at the nodes. Some of the test points are locally pruned, and the remaining ones are then validated at the other nodes by passing them through the ring topology formed by the nodes. The next set of blocks are then read and this process continues until all the test points are exhausted. Unlike the Parallel Bay's algorithm (PBay) [13], our method is expected to be faster since we do not evaluate the same test block in parallel at all the nodes. For a set of test points, we leverage the nearest neighbors found at one node to continue the search on the next node, instead of starting from scratch at all the nodes for that test block. As a result, the size of the test block keeps diminishing as it moves around the ring from one node to the other.

One critical component of any distributed algorithm is the termination criterion. In our algorithm this can be implemented in one of two ways. Each node P_i keeps track of τ_i, the total number of points that it has pruned, and the leader machine keeps track of ρ, the total number of points it received as potential outliers. Periodically the leader polls the workers for their values of τ_i's. Whenever $\sum_{i=1}^{n} \tau_i + \rho = |D|$, the leader sends a terminate message to all the nodes. Alternatively, each node can send a termination signal to the leader when the remaining test block size becomes zero.

The second distributed algorithm that we describe in this paper combines the index developed for *iOrca* and the distributed processing of *DOoR*. We call this the indexed distributed outlier detection algorithm or *iDOoR*. The leader node first reorders the data based on a randomly chosen reference point. This is to ensure that the points which have the highest distance to the reference point (the potential outliers) are processed first. Then it broadcasts the reference

```
Input: D_i, n, k
Output: O_k, the set of outliers
Initialization: O_k ← ∅;
if (x, r_x, L_k(x)) is received then
    ρ ← ρ + 1;
    if |O_k| ≤ t − 2 then O_k ← O_k ⋃ [x; r_x; L_k(x)];
    if |O_k| = t − 1 then
        O_k ← O_k ⋃ [x; r_x; L_k(x)];
        c ← min{δ_k(y, D) : y ∈ O_k};
        Broadcast c to all nodes;
    end
    if |O_k| ≥ t then
        if r_x > min{δ_k(y, D) : y ∈ O_k} then
            Drop y ∈ O_k with minimum δ_k, add x to O_k;
            c ← min{δ_k(y, D) : y ∈ O_k};
            Broadcast c to all nodes;
        end
    end
end
```

Algorithm 4: Distributed outlier detection at master node

point to all the worker nodes and splits the data amongst all the nodes as before. Any worker node on receiving the reference point builds its own index on its local dataset D_i. The rest of the algorithm is similar to *DOoR*. On receiving a block of test points, any worker node executes the spiral search based on the index built on D_i for the entire block. To reduce the number of messages exchanged between the master node and the worker nodes, before every new test block is sent out, the master node checks to see if Lemma 4.1 holds for the test block. Since the index is stored in memory, these computations do not require any extra disk access. If this condition holds for any test block, *iDOoR* can terminate. We omit the pseudo code here due to shortage of space.

It is fairly easy to argue the correctness of *DOoR* or *iDOoR*. First note that if the cutoff $c = 0$, we pass all the data points through all the nodes, thereby ensuring that the nearest neighbors of all the points are correctly found. In the presence of cutoff, the latter is only updated by the leader. At any point during the execution of the algorithm $c_i < c$, guaranteeing no false dismissals.

6. EXPERIMENTS

6.1 Setup

We have implemented our algorithms in Java. The centralized experiments have been run on a 64-bit 2.33 GHz quad core Dell precision 690 desktop running Red Hat Enterprise Linux version 5.4 having 18GB of physical memory. For the distributed experiments, we have used a mid-range linux cluster comprised of 16 slave nodes each containing two, quad-core 2.66GHz CPU totaling 128 cores and 256GB Ram (2Gb/Core). It is controlled by two master nodes and has 30Tb storage. The distributed Java code uses the Java RMI (remote method invocation) framework to mange the worker nodes with a central client handling communication between the distributed threads. In all the experiments, we report the wall time to include both the CPU and I/O time. Note that we do not report accuracy, because our method is guaranteed to produce the same outliers as Orca. We have

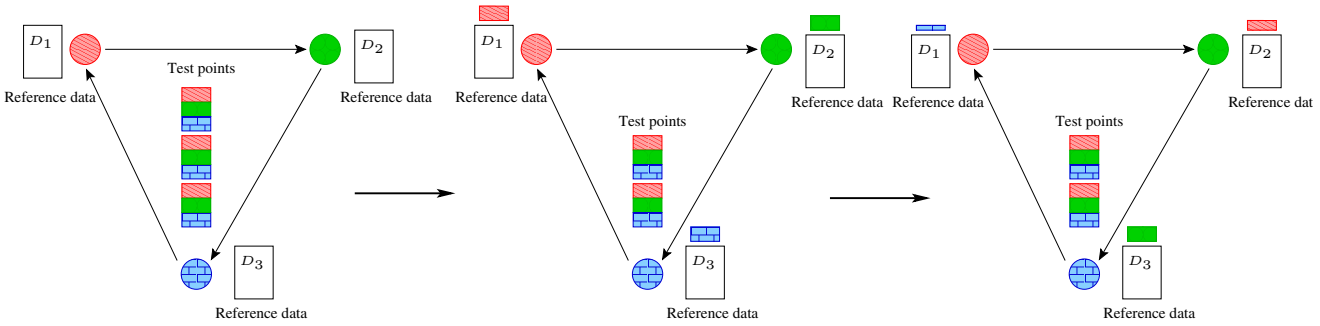
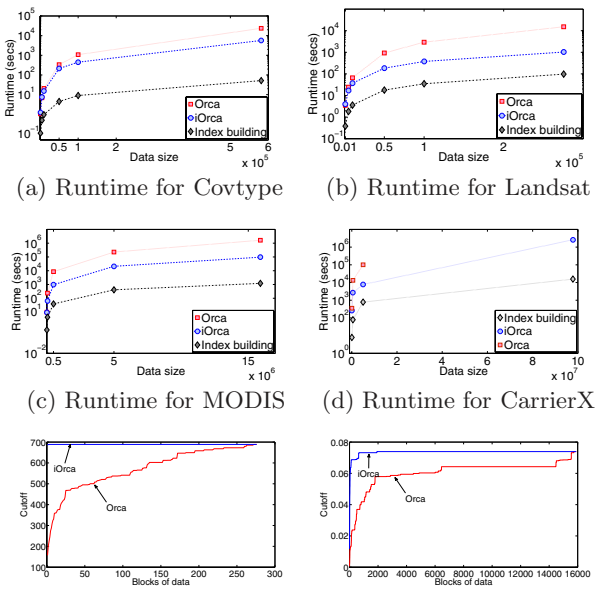


Figure 2: Execution of distributed algorithm. The leftmost picture shows the setup: the test points are color coded to show which block is assigned to which node. Second picture shows that assignment. Third figure shows how the non-pruned points are tested at the other nodes.



(a) Runtime for Covtype (b) Runtime for Landsat

(c) Runtime for MODIS (d) Runtime for CarrierX

(e) Cutoff for Landsat (f) Cutoff for MODIS

Figure 3: Figure showing running time for the 4 datasets ((a) - (d)). As clearly shown for each dataset, *iOrca* beats Orca in running time. This happens because the cutoff increases at faster rate compared to Orca ((e) and (f)).

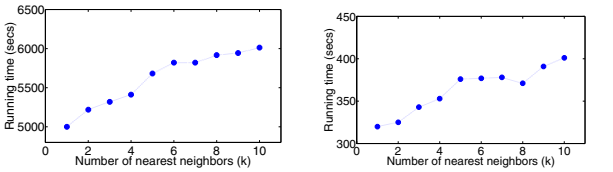
tested our algorithms on the following real datasets[2]:

1. *Covertype*: contains 581,012 instances and 10 features
2. *Landsat*: contains 275,465 instances and 60 features
3. *MODIS*: contains 15,900,968 instances and 7 features
4. *CarrierX*: contains 97,814,864 instances and 19 features

Unless otherwise stated, we have used the following default values for the parameters: block size, $b = 1000$, $t = 30$ and $k = 5$.

[2]The first 2 datasets are available at `http://archive.ics.uci.edu/ml/index.html`. MODIS is available at `http://modis.gsfc.nasa.gov/data/dataprod/dataproducts.php?MOD_NUMBER=09`. The last is a proprietary dataset from a partner airline.

6.2 Centralized results: *iOrca*



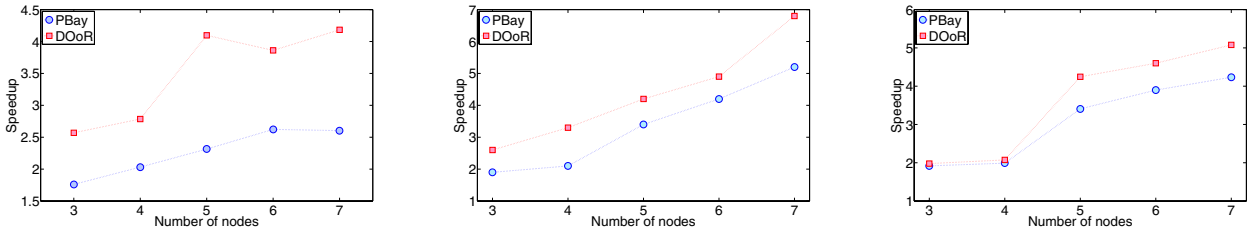
(a) Time vs. k for Covtype (b) Time vs. k for Landsat

Figure 4: Running time vs. k for *iOrca*.

Our first set of experiments demonstrates the benefits of our indexing method in speeding up Orca in a purely centralized setting. Fig. 3 ((a)–(d)) shows the results for the 4 datasets. Each of the Figures (a) - (d) shows the running time (in log scale) along the y-axis for Orca (red squares), *iOrca* (blue circles) and the index building phase of *iOrca* (black diamond). The x-axis are random samples from these datasets. As clearly shown, *iOrca* beats Orca; the decrease in running time is from 2 to 17 times compared to Orca. The decrease in runtime occurs because of the following. (1) The indexing method orders the points in such a manner that those which are farthest from a common reference point are processed first. Since these points are the potential outliers, the cutoff increases much faster (compared to Orca), leading to faster pruning of inliers. (2) The spiral search helps to find the nearest neighbors for each test point much faster than going through the data in the original order. (3) The early stopping criterion helps to avoid testing all the data points. For the Landsat dataset, there are 276 blocks of test data and iOrca only needs to process the first 6 blocks before the early stopping condition signals termination. Similarly, for the MODIS dataset, there are 16,000 blocks of test data and *iOrca* only evaluates the first 13,000 leading to significant savings in both I/O and runtime.

Note that for the CarrierX dataset, we could not plot the runtime of Orca on the entire dataset since it has not finished in more than 8 days. The last two plots (e) and (f) of Fig. 3 compares how the cutoff grows for both Orca and *iOrca* for Landsat and MODIS datasets as each test block is processed.

The next set of figures (Fig. 4) demonstrates the impact on runtime of varying the number of nearest neighbors k for he Covtype and Landsat datasets. As shown, the runtime



(a) Scalability for Covtype (b) Scalability for MODIS (c) Scalability for CarrierX

Figure 5: Speedup of *DOoR* vs PBay [13] algorithm with Orca as the baseline.

increases very slowly as k is increased. This shows that for both these datasets, the spiral search technique is able to find the nearest neighbors of a test point in very little time, leading to a very slow growth in runtime.

Comparison to other methods:

We have compared our algorithm with the RBRP algorithm [9]. Unfortunately, the source code made available by the authors was in C and so we re-implemented it in Java. Overall we found it to be much slower than *iOrca* because of primarily two reasons: (1) the number of clusters produced is roughly of the order of 1000 for most of the datasets; moreover, finding the nearest clusters for each cluster is, in itself, a quadratic time computation with respect to the number of clusters, and (2) as described in Section 2, processing each test point one by one leads to an unacceptably large number ($O(n^2)$) of disk I/O, thereby slowing down the performance dramatically.

6.3 Distributed results: *DOoR* and *iDOoR*

Fig. 5 shows the performance results for *DOoR*. The red curve shows the speedup (ratio of the runtimes) that is obtained by running *DOoR* on 3 to 7 distributed machines when compared against a centralized Orca. The blue curve shows the speedup of obtained by PBay [13] vs. Orca. As shown, the speedup of *DOoR* is much greater compared to that of PBay for all the datasets. For Covertype and MODIS, the speedup is almost linear for *DOoR*. This shows that our distributed algorithm *DOoR* is more efficient in finding outliers compared to PBay. The primary reason for this is that *DOoR* can better utilize the cutoff by assigning different test blocks to different nodes at the beginning, unlike PBay which assigns the same test block to all the nodes. As a result in *DOoR*, each test block keeps shrinking in size as it travels through the ring for validation. Moreover, *DOoR* can leverage the nearest neighbors of the test points found at one node to speed up the search at the successive node in the ring. In contrast, PBay tests all the blocks in parallel at all the nodes, leading to more computations.

Finally, we present the speedup results for *iDOoR* in Fig. 6. In this case, our baseline is the *iOrca* algorithm. We are interested in answering the following question: *Does distributed processing help in reducing the runtime of iOrca*? As shown in the figure, for most cases, *iDOoR* is faster than *iOrca* (speedup > 1 indicates this). The x-axis shows the number of machines used while the y-axis shows the speedup obtained. Each experiment is run for three datasets shown by the three colored bars. Note that in this case, the speedup is less than linear. Due to the index, the cutoff in *iOrca* increases very fast; as a result, most of the outliers are found

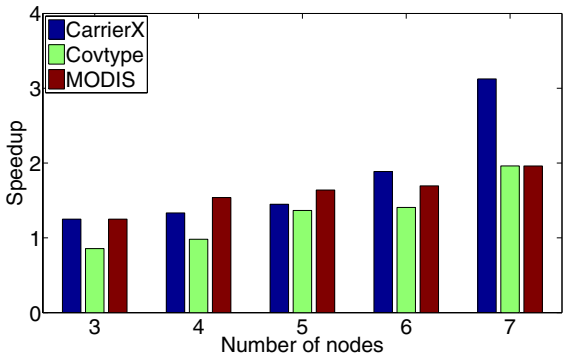


Figure 6: Speedup of *iDOoR* vs. *iOrca*.

in the first 20% of the blocks and the algorithm can process the rest of the data very fast. *iDOoR* helps to speed up the first 20% of the execution, but for the later 80%, of the execution the overhead of message passing is more than the single threaded *iOrca*. This is the prime reason why we do not find a linear increase in speedup when the number of machines is increased. Currently, we are working on a hybrid method whereby we can switch between these two modes on the fly.

7. CONCLUSION

In this paper we have developed both sequential and distributed outlier detection methods that are significantly more efficient than existing methods while still guaranteeing the exact same outliers. Using a simple, but effective indexing and disk block accessing techniques, our sequential algorithm *iOrca* is up to an order-of-magnitude more efficient than the state-of-the-art. By allowing the algorithm to stop early, *iOrca* can terminate even before looking at all the test points. We further propose two distributed algorithms for outlier detection (*DOoR*) and *iDOoR* that can be run on multi-core cluster of machines connected on a ring topology. By combining the index scheme along with distributed processing, our distributed algorithm is much faster than [13] as validated on large public datasets.

Acknowledgments

This research is supported by the NASA SSAT project under NASA Aeronautics Mission Directorate. This work has been approved for public release by The MITRE Corporation, case 11-2367, distribution unlimited. The author's af-

filiation with The MITRE Corporation is provided for identification purposes only, and is not intended to convey or imply MITRE's concurrence with, or support for, the positions, opinions or viewpoints expressed by the author.

8. REFERENCES

[1] F. Angiulli, S. Basta, S. Lodi, and C. Sartori. A Distributed Approach to Detect Outliers in Very Large Data Sets. In *Proocedings of Euro-Par'10*, pages 329–340, 2010.

[2] F. Angiulli and F. Fassetti. DOLPHIN: An Efficient Algorithm for Mining Distance-based Outliers in Very Large Datasets. *ACM TKDD*, 3:4:1–4:57, 2009.

[3] F. Angiulli and C. Pizzuti. Fast Outlier Detection in High Dimensional Spaces. In *Proceedings of PKDD'02*, pages 15–26, 2002.

[4] S. Bay and M. Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *Proceedings of SIGKDD'03*, pages 29–38, 2003.

[5] J. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18:509–517, 1975.

[6] S. Berchtold, D. Keim, and H. Kriegel. The X-tree : An Index Structure for High-Dimensional Data. In *Proceedings of VLDB'96*, pages 28–39, 1996.

[7] M. Breunig, H. Kriegel, R. Ng, and J. Sander. LOF: Identifying Density-based Local Outliers. In *Proceedings of SIGMOD'00*, pages 93–104, 2000.

[8] V. Chandola, A. Banerjee, and V. Kumar. Anomaly Detection: A Survey. *ACM Computing Surveys*, 41:15:1–15:58, 2009.

[9] A. Ghoting, S. Parthasarathy, and M. Otey. Fast mining of distance-based outliers in high-dimensional datasets. *DMKD*, 16:349–364, 2008.

[10] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of SIGMOD'84*, pages 47–57, 1984.

[11] E. Hung and D. Cheung. Parallel Mining of Outliers in Large Database. *Distrib. Parallel Databases*, 12:5–26, 2002.

[12] E. Knorr and R. Ng. Algorithms for Mining Distance-Based Outliers in Large Datasets. In *Proceedings of VLDB'98*, pages 392–403, 1998.

[13] E. Lozano and E. Acuna. Parallel Algorithms for Distance-Based and Density-Based Outliers. In *Proceedings of ICDM'05*, pages 729–732, 2005.

[14] M. Otey, A. Ghoting, and S. Parthasarathy. Fast Distributed Outlier Detection in Mixed-Attribute Data Sets. *DMKD*, 12:203–228, 2006.

[15] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient Algorithms for Mining Outliers from Large Data Sets. In *Proceedings of SIGMOD'00*, pages 427–438, 2000.