Outlier Detection With Streaming Dyadic Decomposition

Chetan Gupta[1,2]
and Robert Grossman[1,3]

[1] Department Of Mathematics, Statistics and Computer Science
University Of Illinois at Chicago
[2] Hewlett Packard Labs
[3] Open Data Group

Abstract. In this work we introduce a new algorithm for detecting outliers on streaming data in $\mathbf{R}^n$. The basic idea is to compute a dyadic decomposition into cubes in $\mathbf{R}^n$ of the streaming data. Dyadic decomposition can be obtained by recursively bisecting the cube the data lies in. Dyadic decomposition obtained under streaming setting is understood as streaming dyadic decomposition. If we view the streaming dyadic decomposition as a tree with a fixed maximum (and sufficient) size (depth), then outliers are naturally defined by cubes that contain a small number of points in the cube itself or the cube itself and its neighboring cubes. We discuss some properties of detecting outliers with streaming dyadic decomposition and we present experimental results over real and artificial data sets.

1 Introduction

Detecting outliers is an important data mining task. In this paper, we are concerned with outlier detection using a streaming data model in which we examine each point once and must detect outliers independent of the stream length.

As streaming data has become more ubiquitous, the problem of detecting outliers on streams has become more important. To give two examples: For example in case of network data, we need to be able to predict the attack patterns in real time over streaming data. In our experimental section we present results over such a data set. Other examples include highway data. If the data is being collected through sensors we need to be able to detect the traffic disruptions in real time as the data streams in.

There are several different definitions of outliers, and quite a few different approaches to detecting them. Some of these are described in the next section.

Roughly speaking, a dyadic decomposition divides $\mathbf{R}^n$ into cubes. A cube may be further divided into sub-cubes, by splitting the edge in each dimension in half. This decomposition produces a collection of dyadic cubes at different scales. It also a produces a tree, which we call a dyadic tree. Each node u of the dyadic tree is associated with a dyadic cube C_u, and a node u is child of another

node v in case the corresponding cube C_u arises by dividing each edge of the cube C_v in half, as described above. A formal definition is given in Section 3.

One natural definition of an outlier is to say a point p in a data set D is an outlier if it is in the cube C_u of a leaf node u. In this case we say that the point p is an *outlier defined by a dyadic decomposition* or ODD. More generally, we can define k-outliers if the cube contains k points or less.

This generalization is useful in identifying members of a minority class.

As a simple example consider a set of points in a 2-D plane. Enclose all the points in a square, divide the square into four smaller squares, such that you could think of them as four quadrants. If all but but one point lie in in Quadrants 1, 2, 3 and only one point lies in Quadrant 4, it is considered an outlier. To present a more trivial example with ordinal variables, consider a set of all tennis players till 1986, with two variables, Wimbledon winners or not and age less than 18 or not. There are lot of players who are less than 18, a lot of players who have won the Wimbledon, but only Boris Becker won it under 18. If the data is plotted in the above manner in a square with four smaller squares, there will be a box with only Boris Becker in it and he is an outlier.

Wherever a large amount of data is streaming in and there is a need to detect anomalous patterns, our approach can prove useful. The other advantage of our approach is that it can handle both the scenarios in a streaming setting: individual outliers or identifying members of a minority class. Our approach is simple, intuitive and works well with both continuous and categorical variables. We have experimented with various data sets and some of the results are presented in the paper. This approach can be extended to identify cluster centers in a streaming setting but due to lack of space we do not present those results here.

The organization of this work is as follows: In Section 2 we give an overview of the existing related work. In Section 3 we define dyadic decompositions. Section 4 we present the algorithm for computing ODDs. Section 5 contains some discussion and Section 6 is experimental studies. Section 6 contains the conclusion.

2 Related Work

Outlier detection is a problem considered in statistics and data mining. In Knorr's [12] work, a point p is defined as an outlier with respect to parameter k and λ, if no more than k points are a distance λ or less from p. In Ramaswamy's [17] work, an outlier is defined as: Given a k and a n, a point p is an outlier if the distance to its $k-th$ nearest neighbor is smaller than the corresponding value for no more than $n-1$ other points. Breunig [4] gives every point a LOF (Local Outlier Factor), which measures how isolated an object is from its surrounding. They assign a degree to every point of being an outlier. They give an example to illustrate the importance of looking at the local rather than the global neighborhood. Since our approach is multiscale in nature, the "global" and "local" change with scale and our approach like [4] takes care of

local and global outliers. The above techniques are not presented for streaming data and ours is a streaming data algorithm.

A dynamic programming approach to finding deviants (outliers in a sense) is presented by Jagdish [10]. Muthu [14] uses this idea to construct a near optimal algorithm in a streaming setting for univariate streams. They extend this as a heuristic for a multivariate setting. Another method in streaming clustering/outlier detection is TECNO-STREAMS [15].

Our approach could be understood as a multi-resolution approach. Wavelets are popular in multi-resolution approaches. WaveCluster [21] is a grid-based clustering method that uses wavelet transform to filter the data. Scale-based clustering has been presented before. One idea is scale space clustering. Chakravarty, in [5] uses Radial Basis Function Network(RBFN) for scale-based clustering. In Wong's [24] work a statistical mechanic-based approach is used where the temperature is the scaling parameter. Another interesting work along similar lines is that of Leung [13] which attempts to provide a unified framework for various scale space approaches. Roberts [18] uses a scale-based smoothing function to estimate probability density function. Our algorithm shows similar properties to these scale-based algorithms. Our way of scaling is different since it is based on dyadic decomposition. Another work is using fractal dimensions is that of Barbara [2].

We can also look at this as a grid based approach. An early grid-based clustering algorithm is that of Warnekar [23]. They address the issue of "neighbors" in a grid setting. Two more grid based techniques are Bang [20] and GRIDCLUST [19] which also create hierarchical clustering using grids. GRIDCLUST is also in a scale-based grid clustering algorithm. Their grid structure is based on a k-d tree. In GRIDCLUST first the cells are arranged in order of their density and merged in that order. Clusters at different scale can be merged also. It is a bottom up approach. Our grid structure is different and we use a top down approach where the idea is to study each scale and see if certain parts of data should be studied at that or a finer scale. DBSCAN [6] is also a similar approach, but it is primarily a clustering algorithm and is not a multiscale dyadic cube approach. We have used data sets from Cure [9] and Chameleon [11]. Again none of these approaches have been presented in context of streaming data.

Recently, clustering algorithms for large data sets and streaming data sets have been developed. BIRCH (Balanced Iterative Reducing and Clustering) [25] clusters large data sets by using specialized tree structures to work with out-of-memory data. CLARANS (Clustering Large Applications based on RANdom Search) [16] identifies candidate cluster centroids through analysis of repeated random samples of the original data. The k-mediod problem is a variation of k-means. Guha, et al. [8] have presented streaming algorithms using local clustering to solve the k-mediod problem. Additional work was done by Bradley, et al. [3] and some of the improvements were made by Farnstrom [7]. Aggarwal [1] presents a method for clustering high dimension data using projections. Outlier detection could be treated as a byproduct of these clustering approaches.

3 Dyadic Decompositions

In this section we provide formal definitions of dyadic decompositions and outliers defined by dyadic decompositions (ODDs). We begin by collecting and restating some standard definitions involving cubes and dyadic decompositions [22].

Definition 1. Dyadic decompositions *are defined recursively as follows. Let the data set be* $D \subseteq \mathbf{R}^n$. *Let* $|D| = N$ *denote the number of points in* D. *First, we enclose the* n*-dimensional data set* D *in a cube. The* scale *of this initial cube is defined to be one. We define additional cubes in our decomposition recursively as follows. For an integer* $k > 1$, *we divide a cube whose scale is* k, *into* 2 *or more cubes whose scale we define to be* $k + 1$, *by bisecting one or more edges of the larger cube. This produces* 2^c *new cubes, where the number of bisections* c *satisfies:* $1 \leq c \leq n$. *We continue this process until a stopping criterion is reached, such as a cube contains fewer than a specified number of points, say,* ϵ, *which may be scale dependent in the sense that* $\epsilon = \epsilon(k)$. *We define this to be* dyadic decomposition.

Definition 2. *The cubes obtained during dyadic decomposition are called* dyadic cubes.

Definition 3. *The dyadic decomposition produces a* dyadic tree, *where each node* u *is associated with cube* C_u, *and a node* u*in the tree is child of another node* v *when the corresponding cube* C_u *arises by dividing one or more edges of the cube* C_v *in half.*

We close this section with a remark:

Remark 1. Note that a cube in 2-dimensions (a square) has 8 possible adjacent cubes, and a cube in 3-dimensions has 26 possible neighboring cubes. In general, there are $3^d - 1$ adjacent cubes for a cube in d-dimensions.

4 Computing the streaming dyadic tree associated with a data set

In a streaming setting since we can look at each point only once and since we have fixed space the tree that we built can only be of a fixed depth. Building a dyadic decomposition as described above requires us to know the whole data set, i.e., before we can divide the cube containing the data set we need to know the dimensions of the cube the data set lies in. This means that we cannot have a top down construction as discussed above. This motivates the construction of streaming dyadic trees.

Let the data stream D be a sequence of points p_1, p_2, ... in $\mathbf{R}^n$. Fix the maximum depth of a tree, an integer, $r_{max} \geq 0$. Let u_0 denote the root of a dyadic tree T and with a slight abuse of notation, let C_{u_0} denote the corresponding

dyadic cube. If u is a node in T, and C_u is the corresponding dyadic cube, let $Count_u$ denote the number of points in the cube C_u.

The idea behind constructing a streaming dyadic tree is simple. Roughly speaking, take the first two points in the stream and build a cube that contains both points. If a new point comes in and it is outside the current cube keep doubling the cube (doubling will result in new cubes) while maintaining the depth of the tree till the new point is in a region covered by the new cubes. This is Step 4. If a new point lies in a region already covered by cubes the point travels to the smallest cube that could contain the point. This is Step 3, Case 1. If the leave is maximum depth increment the count of the leaf and discard the point, this is Step 3, Case 2, otherwise spit the node making a new leaf for the point. This is Step 3, Case3. The precise formulation is presented below.

For given a data stream, the decomposition obtained by the following algorithm is called a *streaming dyadic decomposition*. Associate a tree T with this decomposition.

Algorithm 1: Computing a Streaming Dyadic Tree

1. Let C_0 denote a cube that contains the point p_1. Set $C = C_0$.
2. For $i \geq 2$, score the point p_i using T as follows. Let C_{u_0} denote the cube associated with the root node u_0. Set $C = C_{u_0}$ and proceed to the next step.
3. If p_i is in the cube $C = C_u$ associated with the node u of T, then check each of the following three cases in order:
   - Case 1. In this case, we are at an interior node and will continue processing the point p_i. More precisely, (1) The node u has children, which divide the cube C_u into sub-cubes C_{u_1}, C_{u_2}, ..., with corresponding nodes u_1, u_2, ... and (2) the nodes u_i *are not* at the maximum depth $r_{\max}$. In this case, p_i is in one of the sub-cubes, say C_{u_j}. Set $C = C_{u_j}$, increment the count of the cube C_{u_j} and goto Step 3.
   - Case 2. In this case, we have reached the maximum depth of the tree and we will discard the point p_i. More precisely, (1)The node u has children, which divide the cube C_u into sub-cubes C_{u_1}, C_{u_2}, ..., with corresponding nodes u_1, u_2, ... and (2) the nodes u_i *are* at the maximum depth $r_{\max}$. In this case, p_i is in one of the sub-cubes, say C_{u_j}. Increment the counter associated with the node u_j and discard the point p_i. Goto Step 2.
   - Case 3. In this case, we add a new leaf to the tree containing the point p_i. More precisely, if the cube C has no children (i.e., is a leaf) and the node u *is not* at the maximum depth $r_{\max}$, then split the node u to produce sub-cubes C_{u_1}, C_{u_2}, ... Note that in this case, by the construction, the node C_u contains a least one point prior to the arrival of the point p_i. Also note, as mentioned, that the new leaf contains the point p_i. Goto Step 2.
4. Else, if p_i is not in the C_{u_0}, double the cube C_{u_0} until the point p_i is contained in it. Each time, the root is doubled, maintain the maximum depth of T by merging all the leaves with their parents as required. Process the point p_i, by letting $C = C_{u_0}$ and going to Step 3.

The tree obtained in the above algorithm is called a *streaming dyadic tree* associated with a data set D.

4.1 Outliers Associated with Streaming Dyadic Decompositions

Using the dyadic tree associated with a stream, we can now define outliers.

Let D be a data stream and T_{sD} the associate streaming dyadic tree.

Definition 4. *A* outlier associated with a streaming dyadic decomposition or ODD *is a point that i) is contained in cube C_u associated with a leaf node u of the tree T_{sD}; and ii) the cube C_u contains precisely one point. We say the ODD is of depth d in case the leaf C_u is at depth d from the root.*

The above definition can be extended to a *k-outlier associated with a dyadic decomposition or k-ODD* by stipulating that the cube C_u contain at-most k points.

This concept can prove useful in certain practical problems where two or more points are close to each other and away from rest of the points.

Our approach has two key ideas:

1. In our approach we do a streaming division of space into dyadic cubes. If a point lies outside the currently bounded space we can keep doubling the space (while maintaining the depth of the tree) along all the dimensions till the point is inside the bounds. If it lies inside the bounds we either find a path to a leaf or we can keep subdividing the cubes till either the point lies in its own cube or the maximum depth is reached and further subdivision is not permissible.
2. Since a streaming setting means a restriction of space we save space by storing just those points that are possible ODDs. That is why only leaves of the streaming dyadic tree, T_{sD}, can store points. All points which are stored in the leaves are candidate ODDs. At the end of the stream we just need to look at the leaves containing point(s)to obtain our list of ODDs.

We close this section with few remarks:

Remark 2. We can relate this definition of an outlier to Knorr's definition of an outlier. A point p is an defined as an *Knorr outlier* with respect to parameters k and λ, if no more than k points are a distance λ or less from p. We see that ODDs at depth d are similar in spirt to Knorr outliers but have a natural scale associated with them and use (empty or sparse) dyadic decompositions at scale d instead of a distance λ.

Remark 3. We can put restrictions on the space in which a point can exist inside a dyadic cube. This can result in more restrictive definition of an ODD.

Remark 4. The depth of the tree needs to be fixed in advance. It is easy to see that more the depth more nodes are available in the tree, increasing the number of candidate ODDs.

4.2 Complexity

The algorithm to building a streaming dyadic tree is designed to work for streaming data. Hence the space and time requirements are severe. Let the tree contain a total N_t nodes and leaves at any time t. N_t is bounded by fixing the maximum depth of the tree. The worst case scenario for any operation is traversing the complete tree for a worst case complexity of $O(N_t)$.

The space required is to store these nodes, $O(N_t)$ and the candidate ODDs.

Pruning is an effective solution for saving time and space. The tree can be pruned without losing any information. Once all the children of a particular node contain more than one element that node can be considered full. If a point falls within the bounds of that node it cannot be an outlier and it is discarded. This obviously saves the computation involved in further travelling down that branch of the tree. This does not affect the accuracy of the results.

In our experiments, even with a data sets of two million points in eight dimensions space was never an issue since typically, the data clusters naturally to a few branches of the tree.

5 Experimental Results

We have completed some experimental studies on artificial and real data sets using the algorithm described above.

The only variable the user needs to input is the maximum depth of the tree. We want to see how many outliers we are able to identify as ODDs.

To test the algorithm we did four series of experiments.

1. 2-D data sets: The first series of experiments used synthetic 2-D data sets.
2. The second series of experiments used large synthetic 2-D data sets.
3. The third series used data set containing computer network alerts.
4. The fourth series used data sets of NASA shuttle data.

For some experiments we have computed two measures, *Sensitivity* and *Specificity* and their sum *Goodness*.

$$Sensitivity = \frac{True\ Positives}{Total\ Positives} \tag{1}$$

$$Specificity = 1 - \frac{False\ Positives}{Data\ Set\ Size} \tag{2}$$

$$Goodness = Sensitivity + Specificity \tag{3}$$

Note that declaring all the points as outliers gives *sensitivity* as one but *specificity* is almost zero making the total *goodness* almost one. Not declaring any point as an outlier makes *specificity* one but the *sensitivity* is zero, making the *goodness* again one. Randomly declaring any point as an outlier with probability half also leads to a *goodness* of one. Therefore, any improvement over one is an

improvement in terms of outlier detection. If an algorithm is indiscriminate in labelling points as outliers, and in the process wrongly labels a lot of points as outliers, the *sensitivity* might be high but the *specificity* goes down. On the other hand if an algorithm declares too few points as outliers, false positives would decrease and *specificity* will be high but the *sensitivity* will go down. In our experiments, as the maximum value for the depth of the tree r_{max} is increased the *sensitivity* goes up but the *specificity* goes down.

The algorithm is designed for a streaming setting. To simulate a streaming setting we read one record at a time from a file stored on a drive and discarded the point after that. Some of the real data sets that we have used contain categorical variables. We converted them to a set of binary variables using the simple technique of converting each category as yes/no value and adding it as an attribute.

5.1 2-D Data Sets

These are the Cure [9] and Chameleon [11] data sets which have non-spherical clusters and some outlier points distributed through the plane. In Figure 5.1(Please see the last page) for the three individual figures, we have colored points identified as ODDs with the color blue. It is easy to see from the figures that in general what would appear as an outlier to the naked eye is also identified as an ODD by our algorithm. The first three data sets have 8000 points and the last one has 10000 points.

5.2 Large Data Sets

We created 30 large synthetic data sets. They were of three types, consisting of ten sets each:

1. The first type was in four dimensions and had $100,000$ points. The data was distributed in four clusters and $1, \ldots, 10$ percent of the points (meaning 1000 to $10,000$) were distributed randomly to create ten data sets respectively. The random points are expected to be the outliers our algorithm identifies as ODDs.
2. The second type was in four dimensions and had $1,000,000$ points. The data was distributed as with the previous experiment.
3. The third type was in eight dimensions and had $2,000,000$ points. The data was distributed as previously but with five clusters.

Results In Table 1 we have tabulated these results. True positives indicate the number of correct labels, i.e., the number of outliers that were identified as ODDs by our algorithm and the false negatives indicate the number of outliers that the algorithm failed to identify as ODDs.

For the majority of the experiments, the sensitivity was greater than 0.99. In other words, the algorithm identifies almost all outliers as ODDs. For the

majority of the experiments, the specificity was greater than 0.99 and the picks included very few false negatives, i.e., those outliers that the algorithm failed to identify as ODDs.

Number Points	Noise %	True Positives	False Negatives	Sensitivity	Specificity	Goodness
$0.1 * 10^6$	1	924	76	0.924	0.99995	1.92395
$0.1 * 10^6$	2	1903	97	0.9515	0.99994	1.95144
$0.1 * 10^6$	3	2957	42	0.9859	0.99979	1.98569
$0.1 * 10^6$	4	3999	1	0.9997	0.99881	1.99851
$0.1 * 10^6$	5	4988	12	0.9976	0.99976	1.99736
$0.1 * 10^6$	6	5996	3	0.9994	0.99866	1.99806
$0.1 * 10^6$	7	6721	279	0.9601	0.9999	1.96
$0.1 * 10^6$	8	7988	12	0.9985	0.99949	1.99799
$0.1 * 10^6$	9	8992	8	0.9991	0.99867	1.99777
$0.1 * 10^6$	10	9977	23	0.9977	0.99981	1.99751
$1.0 * 10^6$	1	9999	1	0.9999	0.99952	1.99942
$1.0 * 10^6$	2	19991	9	0.9995	0.99763	1.99713
$1.0 * 10^6$	3	29991	9	0.9997	0.99737	1.99707
$1.0 * 10^6$	4	39930	69	0.9982	0.99988	1.99808
$1.0 * 10^6$	5	49404	596	0.9881	0.99998	1.98808
$1.0 * 10^6$	6	59887	111	0.9981	0.9999	1.998
$1.0 * 10^6$	7	69678	320	0.9954	0.99997	1.99537
$1.0 * 10^6$	8	79935	65	0.9992	0.99985	1.99905
$1.0 * 10^6$	9	89473	527	0.9941	0.99992	1.99402
$1.0 * 10^6$	10	99711	286	0.9971	0.99984	1.99694
$2.0 * 10^6$	1	19827	91	0.9954	0.9999	1.9953
$2.0 * 10^6$	2	39564	226	0.9943	0.9998	1.9941
$2.0 * 10^6$	3	56343	3657	0.9391	0.9999	1.939
$2.0 * 10^6$	4	79979	21	0.9997	0.9999	1.9996
$2.0 * 10^6$	5	99999	1	0.9999	0.9971	1.997
$2.0 * 10^6$	6	11999	1	0.9999	0.9993	1.9992
$2.0 * 10^6$	7	139044	956	0.9931	0.9999	1.993
$2.0 * 10^6$	8	158687	1313	0.9917	0.9999	1.9916
$2.0 * 10^6$	9	17998	2	0.9999	0.8731	1.873
$2.0 * 10^6$	10	19981	19	0.9999	0.9995	1.9994

Table 1. Results of finding ODDs in several large artificial data sets

5.3 KDD Data: Network Alert Data

We did experiments using a KDD-99 data set [http://kdnuggets.org/] to create several data sets to test our algorithm. Since it can be tricky to identify a point a-priori as an outlier in a data set, we used the KDD labels. The data set had a total 311,029 records with 37 different types of network attack patterns and records labelled "normal". We picked one record each at random from every attack

type and randomly picked either $\{1000, 5000, 10000, 25000, 50000\}$ of the normal patterns for a total of $\{1037, 5037, 10037, 25037, 50037\}$ points respectively.

We ran our algorithm on five data sets for each size (for a total of 25 data sets) and for a maximum depth, r_{max} of six to ten. We tabulate the results in Table 2. Due to lack of space, for each data set we have chosen one sample run. We have tabulated the depth, r_{max}, number of correct outlier labels, sensitivity, specificity and goodness.

Results In all our runs more than half the attack patterns were flagged as ODDs by our algorithm. Notice that except for one, all of the goodness values are above 1.5 and most of them are greater than 1.6.

Typically, as r_{max} is increased the number of points flagged as ODDs increases. This would mean that sensitivity, which measures true positives, goes up but false positives go up too, reducing the specificity. In Table 2, for all experiments more true outliers could have been flagged as ODDs than the number indicated, but it would have resulted in a loss of goodness. Moreover, the attack types that the algorithm failed to identify were invariably almost the same over all the different experiments.

5.4 Identifying a Minority Class

In the experiments just described, streaming ODDs were defined by using leaves in the trees that contain a single point. In the next series of experiments, we relaxed this restriction and let the leaves contain k or fewer points. This time we also picked k attack patterns for every attack type.

Experiment 1 The results are summarized in Table 3, where k is called the threshold. This time all of the goodness values are above 1.5 and again most of them are greater than 1.6.

Experiment 2 In another experiment with the network data set, we picked all the examples of those attack types that had less than 25 examples. In total there were 166 attack patterns. There were 20 such attack types. We also randomly picked 5000 points. We tried various threshold cardinalities.

The best result (in terms of goodness) was for $k = 7$ of a k-ODD, which gave goodness of 1.59 and 121 of 166 attack patterns were identified. For $k = 9$ we were able to capture 146 of 156 attack patterns, but the specificity was low.

The higher cardinality works because (1) Some patterns of the same attack types are very similar. (2) Different attack types sometimes also have similar patterns. (3) Some normal patterns occur with the attack types.

5.5 NASA Shuttle Data

This data has seven classes and 14,500 trained examples in nine dimensions [http://kdnuggets.org/]. There are 11,478 examples of class one, 12 examples of

Number Points	Depth	True Positive	Sensitivity	Specificity	Goodness
1000	6	24	0.648648649	0.94021215	1.588860799
1000	9	33	0.891891892	0.753134041	1.645025932
1000	6	22	0.594594595	0.941176471	1.535771065
1000	7	30	0.810810811	0.849566056	1.660376867
1000	6	22	0.594594595	0.947926712	1.542521306
5000	8	27	0.72972973	0.900933095	1.630662825
5000	9	28	0.756756757	0.848520945	1.605277702
5000	10	33	0.891891892	0.786182251	1.678074143
5000	7	30	0.810810811	0.922771491	1.733582302
5000	9	33	0.891891892	0.805042684	1.696934576
10000	8	25	0.675675676	0.926472053	1.602147729
10000	9	28	0.756756757	0.870279964	1.627036721
10000	9	27	0.72972973	0.89628375	1.62601348
10000	10	31	0.837837838	0.786191093	1.624028931
10000	10	28	0.756756757	0.861313141	1.618069898
25000	9	27	0.72972973	0.880257219	1.609986949
25000	8	29	0.783783784	0.90142589	1.685209673
25000	10	26	0.702702703	0.89603387	1.598736573
25000	10	27	0.72972973	0.885809003	1.615538732
25000	10	29	0.783783784	0.86995247	1.653736254
50000	10	23	0.621621622	0.87401323	1.495634852
50000	9	30	0.810810811	0.937706097	1.748516908
50000	10	23	0.621621622	0.916841537	1.538463159
50000	9	26	0.702702703	0.921737914	1.624440617
50000	10	27	0.72972973	0.893478826	1.623208555

Table 2. Results of finding ODDs in network alert data

Points	Threshold	ODDs	True Positive	Sensitivity	Specificity	Goodness
5000	2	73	54	0.739726027	0.8928	1.632526027
5000	3	103	69	0.669902913	0.9084	1.578302913
5000	4	132	105	0.795454545	0.8156	1.611054545
5000	5	160	114	0.7125	0.909	1.6215
10000	2	73	53	0.726027397	0.8954	1.621427397
10000	3	103	77	0.747572816	0.9022	1.649772816
10000	4	132	92	0.696969697	0.903	1.599969697
10000	5	160	119	0.74375	0.8818	1.62555
25000	2	73	51	0.698630137	0.90832	1.606950137
25000	3	103	83	0.805825243	0.84108	1.646905243
25000	4	132	98	0.742424242	0.9188	1.661224242
25000	5	160	104	0.65	0.94492	1.59492

Table 3. Results of finding those outliers in network alert data that might not occur as singletons

class two, 38 examples of class three, 2155 examples of class four, 807 examples of class five, 4 examples of class six and 2 of class seven.

Our algorithm identifies all 6 points belonging to class six and seven as ODDs in a list of 36 ODDs.

5.6 Building Forests

Instead of using single trees to find streaming ODDs trees we can also use forests of trees to compute streaming CDDs . In Table 4 we have presented some results with one technique for creating forests. The data set consisted of 2 million points with outliers varying from 1-10%.

Results We have tabulated the depth at which both specificity and sensitivity is greater that 0.99 for the first time (with noise of 9%, the forest approach could not reach the level of 0.99 sensitivity). It can be seen though that for most experiments the depth of a forest is less than that of a single tree.

Outlier Percent	Single Tree	Forest
1	9	7
3	10	9
4	9	7
5	7	7
6	8	8
7	10	10
8	11	9
9	7	-
10	9	7

Table 4. Maximum depth needed to achieve greater that 0.99 specificity and sensitivity for streaming ODDs for a data set with 2×10^6 points and outlier varying form $1 - 10\%$ using single tree and forest

A number of approaches can be used to create a forest. Our idea is simple:

1. Insert a new point in the tree within whose root's bounds the point falls.
2. If no such tree exists:
   (a) If a tree has less than maximum depth, insert the point in that tree.
   (b) Else create a new tree.
   (c) Once the number of trees is equal to maximum number of allowable trees in the forest, insert the new point in a tree that will require the least doubling to accommodate the point within its bounds.

5.7 Discussion

We have conducted experiments with two artificial and and two real life data sets.

The only variable we need to fix to run the algorithm is to fix the maximum depth of the dyadic tree.

The maximum depth of the tree determines the number of points that are identified as ODDs. More the depth, greater the number of points identified as ODDs. In our experiments a depth of 6-10 seems to have worked well. The same holds true for identifying member of a minority class.

One possible drawback of our approach is sparse data, in which it is difficult to define what an outlier is.

6 Conclusions

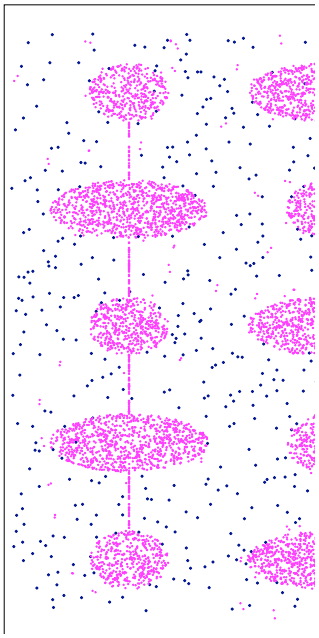
In this work, we have introduced a streaming algorithm for detecting outliers that is simple, effective and naturally exploits the multiscale nature of many common data sets. It is based upon a natural modification of a dyadic decomposition of a data set when the data is presented in the form of a stream and only a finite amount of space is available to construct the dyadic tree.

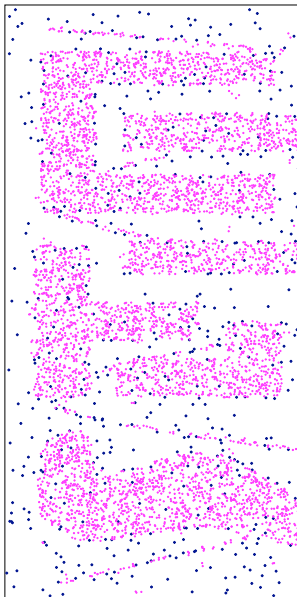
Once we have constructed a dyadic tree under a streaming setting we use it to find outliers in a streaming setting. We have described a few modifications to our approach, e.g., forests, k-ODDs. Finally, we have conducted experiments on artificial and real data sets to demonstrate the use of our algorithm.

References

1. C. C. Aggarwal, J. Han, and P. S. Yu. A framework for projected clustering of high dimensional data streams. *Proceedings of the 30th VLDB Conference*, 2004.
2. D. Barbara and P. Chen. Using fractal dimension to cluster data sets. *In Proceedings of the* 6^{th} *ACM SIGKDD*, pages 260–264, 1999.
3. P. S. Bradley, U. M. Fayyad, and C. A. Reina. Scaling clustering algorithms to large databases. *In proceedings of the 4th Int'l Conference on Knowledge Discovery and Data Mining*, pages 9–15, Aug 27-31, 1998.
4. M. M. Breunig, H. Kriegel, R. T. Ng, and J. Sander. Lof: Identifying density based local outliers. *In Proceedings of the ACM International Conference Management Of Data*, pages 93–104, 2000.
5. S. V. Chakravarty and J. Ghosh. Scale based clustering using the radial basis function network. *IEEE Transcations on Neural Networks*, 1996.
6. M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. *In Proc. of the Second Intl Conference on Knowledge Discovery and Data Mining*, 1996.
7. F. Farnstrom, J. Lewis, and C. Elkan. True scalability of clustering algorithms. *SIGKDD Explorations*, 2000.
8. S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. *In the Annual Symposium on Foundations of Computer Science, IEEE*, 2000.
9. S. Guha, R. Rastogi, and K. Shim. Cure: An efficient clustering algorithm for large databases. *Proc. of 1998 ACM-SIGMOD International Conference on Management of Data*, 1998.

10. H. V. Jagdish, N. Koudas, and S. Muthukrishnan. Mining deviants in a time series data base. *Proceedings of International Confernce on Very Large Databases, VLDB '99*, 1999.
11. G. Karypis, E.-H. Han, and V. Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *IEEE Computer*, 32(8):68–75, 1999.
12. E. Knorr and R. Ng. Algorithms for mining distance based outliers in large datasets. *Proceedings of International Conference on Very Large Databases, VLDB '98*, pages 392–402, 1998.
13. Y. Leung, J.-S. Zhang, and Z-B Xu. Clustering by scale-space filtering. *IEEE Transactions on Pattern Analysis And Machine Intelligence*, 22(12), December 2000.
14. S. Muthukrishnan, R. Shah, and J. S. Vitter. Mining deviants in time series data streams. Technical Report DIMACS TR 2003-43, DIMACS, 2003.
15. O. Nasraoui, C. C. Uribe, C. R. Coronel, and F. Gonzalez. Tecno-streams: Tracking evolving clusters in noisy data streams with a scalable immune system learning model. *Third IEEE International Conference on Data Mining (ICDM'03)*, 2003.
16. R. Ng and J. Han. Very large data bases. *In Proceedings of the 20th International Conference on Very Large Data Bases, VLDB '94*, pages 144–155, September 1994.
17. S. Ramaswamy, R. Rastogi, and K. Shim. Efficient algorithms for mining outliers from large data sets. *In Proceedings of the ACM International Conference Management Of Data,SIGMOD '00*, pages 427–438, 2000.
18. J. S. Roberts. Parametric and non-parametric unsupervised cluster analysis. *Pattern Recognition*, 30(2):261–272, 1997.
19. E. Schikuta. Grid clustering: A fast hierarchical clustering method for very large data sets. *In Proceedings 13th International Conference on Pattern Recognition*, 2:101–105, 1996.
20. E. Schikuta and M. Erhart. The bang clustering system: A grid based data analysis. *In Proceedings Advances in Intelligent Data Analysis, Reasoning About Data*, 2^{nd} International Symposium:513–524, 1997.
21. G. Sheikholeslami, S. Chatterjee, and A. Zhang. Wavecluster: A multi-resolution clustering approach to very large databases. *Proceedings of the* 24^{th} *VLDB Conference, VLDB '98*, 1998.
22. E. M. Stein. *Singular Integrals and Differentiability Properties of Functions*. Princeton University Press, 1970.
23. C. S. Warnekar and G. Krishna. A heuristic clustering algorithm using union of overlapping pattern cells. *Pattern Recognition*, 11:85–93, 1979.
24. Y. F. Won. Clustering data by melting. *Neural Computation*, 5(1):89–104, 1993.
25. T. Zhang, R. Ramkrishnan, and M. Linvy. Birch: An efficient data clustering method for very large databases. *SIGMOD*, 25(2):103–114, 1996.





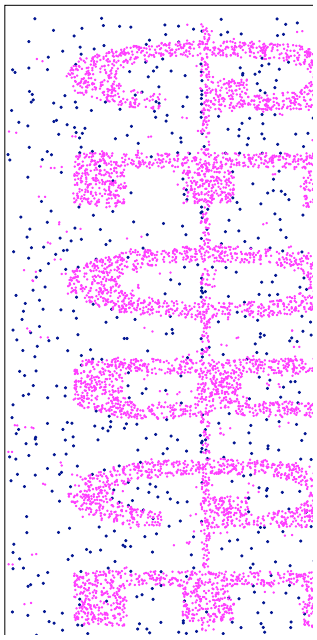


Fig. 1. Visual description of results of outlier detection for a data set drawn from Cure/Chameleon data sets