Continuous Outlier Detection in Data Streams: An Extensible Framework and State-Of-The-Art Algorithms

D. Georgiadis
Aristotle University
Thessaloniki, Greece
dkgeorgi@csd.auth.gr

M. Kontaki
Aristotle University
Thessaloniki, Greece
kontaki@csd.auth.gr

A. Gounaris
Aristotle University
Thessaloniki, Greece
gounaria@csd.auth.gr

A. N. Papadopoulos
Aristotle University
Thessaloniki, Greece
papadopo@csd.auth.gr

K. Tsichlas
Aristotle University
Thessaloniki, Greece
tsichlas@csd.auth.gr

Y. Manolopoulos
Aristotle University
Thessaloniki, Greece
manolopo@csd.auth.gr

ABSTRACT

Anomaly detection is an important data mining task, aiming at the discovery of elements that show significant diversion from the expected behavior; such elements are termed as outliers. One of the most widely employed criteria for determining whether an element is an outlier is based on the number of neighboring elements within a fixed distance (R), against a fixed threshold (k). Such outliers are referred to as *distance-based* outliers and are the focus of this work. In this demo, we show both an extendible framework for outlier detection algorithms and specific outlier detection algorithms for the demanding case where outlier detection is continuously performed over a data stream. More specifically: i) first we demonstrate a novel flavor of an open-source publicly available tool for Massive Online Analysis (MOA) that is endowed with capabilities to encapsulate algorithms that continuously detect outliers and ii) second, we present four online outlier detection algorithms. Two of these algorithms have been designed by the authors of this demo, with a view to improving on key aspects related to outlier mining, such as running time, flexibility and space requirements.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*data mining*

General Terms

Algorithms, Performance

Keywords

outlier detection, continuous processing, data streams, metric space

1. INTRODUCTION

Outlier mining is considered an important task in many applications, such as fraud detection, plagiarism, computer network management, event detection (e.g., in sensor networks), to name a few. An object is characterized as outlier if it does not show the expected behavior, which means that it corresponds either to noise or to important knowledge. In both cases, these deviating objects must be detected and reported.

In this work, we focus on *distance-based* outliers [6], where objects belong to a metric space, and, thus, the distance function used satisfies the triangular inequality. According to this definition, an object x is marked as outlier, if there are less than k objects located at a distance at most R from x. Figure 1 illustrates an example, where object b is an outlier for k=3, since there are less than 3 objects in the R-neighborhood of b. The rest of the objects are marked as inliers, because there are at least 3 objects in their R-neighborhood.

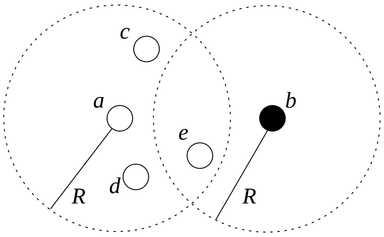


Figure 1: Distance-based outlier example for k=3.

We are interested in outlier detection over *data streams* [1], where new objects are continuously arrive whereas old ones expire. We follow the *count-based sliding window* approach, where each time a new object arrives the oldest one expires, thus, keeping the set of *active objects* constant. The set of active objects is organized by means of a metric-based access method, e.g., an M-tree [5], to facilitate efficient range query processing; range query processing is the main approach to computing the number of objects in the R-neighborhood of an object.

Mining data streams is more challenging than mining a static set of objects, mainly because of the dynamic nature of the objects. In our case, an object may change its status through its lifetime. For example, an object x that is an outlier at time t_1 may lose this property at time t_2 and become and inlier. Evidently, there may be objects that retain the same state (either inlier or outlier) from the time they become active until the time they expire.

The goal of this demonstration is two -fold:

1. to perform a comparison of the state-of-the-art algorithms for continuous outlier detection over data streams using the sliding window approach, and
2. to implement these algorithms in the MOA system [3], thus extending its capabilities further, beyond classification and clustering tasks which are currently supported.

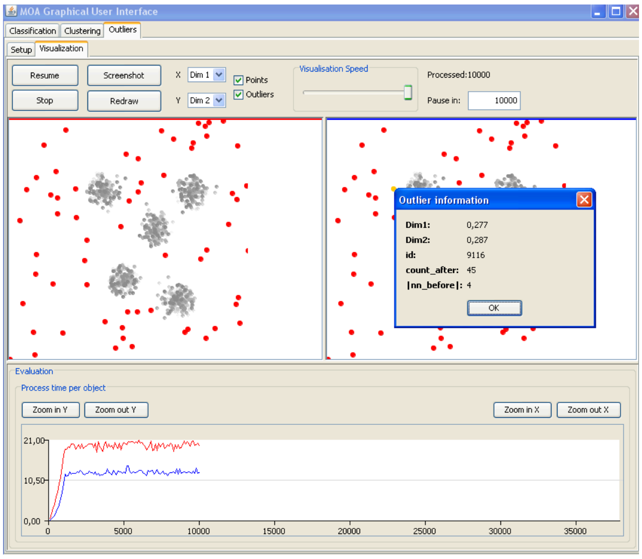
There are four algorithms that have been implemented and compared: STORM [2], Abstract-C [8], COD [7] and MCOD [7]. We note that algorithms COD and MCOD have been designed by the authors of this demonstration and they are studied in detail in [7]. The rest of the work is organized as follows. In the next section we describe briefly the extensions that we performed in MOA toward supporting continuous outlier detection. Section 3 presents the algorithms under study, whereas Section 4 describes the demonstration scenarios that will be considered. Finally, Section 5 concludes the work and discusses briefly some future directions.

2. EXTENDING THE MOA FRAMEWORK

MOA is an open source, publicly available[1] data mining tool that builds upon the work in WEKA[2] [3]. Its main purpose is to provide an easily extensible framework for implementing, evaluating and benchmarking classification and clustering algorithms tailored to streaming scenarios. The MOA system architecture revolves around a three-phase setting process that is common to any data mining task: i) first, a data stream is defined and configured; ii) second, the classification or clustering algorithm is selected and configured; and iii) finally, the evaluation method or measure is chosen. After this setup, which also includes the definition of the size of the sliding window, streaming data mining tasks are ready to run and produce results. Each of the above three phases is extensible.

The data mining tasks currently supported by MOA include stream classification and clustering. Although there exist clustering algorithms that are capable of producing the set of outliers as a by-product (e.g., [4]), MOA does not currently support stream outlier detection as a stand-alone function. Our extensions aim to fill this gap. We started by creating a new tab in the tool that is devoted to outlier detection going beyond the suggested MOA extensibility points. We largely built upon the code for the clustering tab and we created a new package, namely `moa.gui.outliertab`, as shown in Figures 2 and 3. This is the first step towards rendering outlier detection a first class citizen in MOA.

The second phase in the extensions aimed to allow for uniform implementation of stream outlier detection algorithms. Again, we tried to reuse the code in clustering to the largest possible extent. Outlier detection algorithms inherit an abstract class called `MyBaseOutlierDetector`, which extends `AbstractClusterer`. According to that, each outlier detection algorithm basically implements three methods:

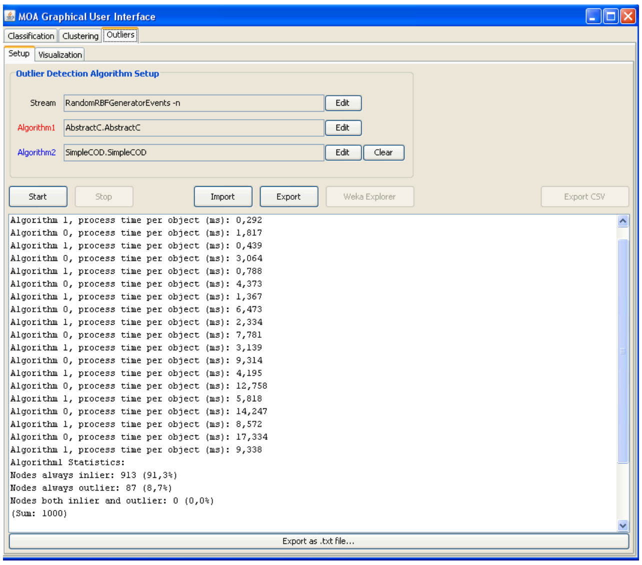


Processed:10000

Pause in: 10000

Dim1: 0,277
Dim2: 0,287
id: 9116
count_after: 45
|nn_before|: 4

Figure 2: The new outlier functionality in MOA.



Stream: RandomRBFGeneratorEvents -n
Algorithm1: AbstractC.AbstractC
Algorithm2: SimpleCOD.SimpleCOD

```
Algorithm 1, process time per object (ms): 0,292
Algorithm 0, process time per object (ms): 1,817
Algorithm 1, process time per object (ms): 0,439
Algorithm 0, process time per object (ms): 3,064
Algorithm 1, process time per object (ms): 0,788
Algorithm 0, process time per object (ms): 4,373
Algorithm 1, process time per object (ms): 1,367
Algorithm 0, process time per object (ms): 6,473
Algorithm 1, process time per object (ms): 2,334
Algorithm 0, process time per object (ms): 7,781
Algorithm 1, process time per object (ms): 3,139
Algorithm 0, process time per object (ms): 9,314
Algorithm 1, process time per object (ms): 4,195
Algorithm 0, process time per object (ms): 12,758
Algorithm 1, process time per object (ms): 5,818
Algorithm 0, process time per object (ms): 14,247
Algorithm 1, process time per object (ms): 8,572
Algorithm 0, process time per object (ms): 17,334
Algorithm 1, process time per object (ms): 9,338
Algorithm1 Statistics:
Nodes always inlier: 913 (91,3%)
Nodes always outlier: 87 (8,7%)
Nodes both inlier and outlier: 0 (0,0%)
(Sum: 1000)
```

Figure 3: Detailed statistics gathering screenshot.

- *void Init()*, which initializes the algorithm;
- *void ProcessNewStreamObj(Instance inst)* , which adds a new object in the current window and, if the window is full, it discards the oldest object[3];
- *Vector<Outlier> getOutliersResult()*, which returns the current set of outliers either for visualization or for evaluation.

In Section 3, we discuss more thoroughly the exact state-of-the-art algorithms that have already been implemented.

In the final step, we extended the `moa.gui.visualization` package for visualizing outliers. The visualization extensions are responsible for drawing outliers and inliers in different

[1] from `http://moa.cs.waikato.ac.nz/`

[2] `http://www.cs.waikato.ac.nz/ml/weka/`

[3] At the moment, the implementation considers only count-based sliding windows.

colors and to support mouse click events on outliers so that they report the number of preceding and succeeding neighbors (see Figure 2). Moreover, the system is capable of reporting performance statistics related to the evolving average processing time per data object and the number of objects that have been outliers or inliers throughout their lifetime, or have changes status at some point in their lifetime (see bottom panel in Figure 2 and Figure 3).

3. ALGORITHMICS

Continuous outlier detection is a special class of steam data mining. Typically, stream data mining algorithms assume that each object is inspected at most once. However, in continuous outlier detection we need to be capable of reporting, at each time point, the outliers among all the objects in the current sliding window. This entails that we need to continuously inspect each object that has not expired (either directly or indirectly) rather than inspecting it only once (e.g., when it arrives). The reason is that an object may change its outlierness status during its lifetime. This characteristic aggravates the need for high time and space efficiency.

The criteria according to which an object is classified as an outlier may vary across different techniques. In this work, we focus on *distance-based* outliers, which capture a broad range of scenarios: Given two parameters, $R \geq 0$ and $k \geq 0$, a distance-based outlier is every object that has less than k neighbors in distance at most equal to R [6]. Moreover, we are mostly interested in exact algorithms. In the sequel, we will briefly describe four distance-based continuous outlier algorithms that we are also going to demonstrate. We will present them in chronological order of their first proposal.

3.1 STORM and Abstract-C

A naive solution to the problem of continuous detection of distance-based outliers over windowed data streams would involve keeping for each object the complete set of its neighbors. Clearly, such an approach is characterized by quadratic space requirements in the worst case; as such, it is practically infeasible for large windows. Two more efficient approaches to this problem are as follows. According to STORM [2], for each object p, it is sufficient to keep at most k preceding neighbors and just the number of its succeeding neighbors to detect the distance-based outliers for specific R and k. Furthermore, for each new object, a range query with radius R is executed to determine the new object's neighbors, which are then added to its list of preceding neighbors. Furthermore, for each such neighbor, its number of succeeding neighbors is increased by one. At any time instance, the approach adopted by [2] to decide if an object p is an outlier involves the computation of the objects in the list of preceding neighbors that have not expired yet. This cost is $O(logk)$, which means that the cost for all objects is $O(nlogk)$.

The Abstract-C approach in [8] reduces this cost to $O(n)$, as it continuously keeps the number of neighbors of an object for all window slides until its expiration. Because of that, the approach in [8] has worst case space requirements $O(n * windowsize)$, as it maintains as many counters for each object as the window slides for which this object is active. In the worst case, the space requirements can become equal to $O(n^2)$. Moreover, each of these counters may be updated multiple times before becoming obsolete. However, [8] can answer queries with multiple values of k, and works for both time-based and count-based sliding windows. The extended version of MOA to be demonstrated supports the two approaches mentioned above and, additionally, the approximate flavor of STORM.

3.2 COD and MCOD

COD and MCOD have been recently proposed by the authors in order to combine the best of the time and space complexities of STORM and Abstract-C, and also to significantly improve average case performance through the vast reduction of the amount of the costly range queries performed [7].

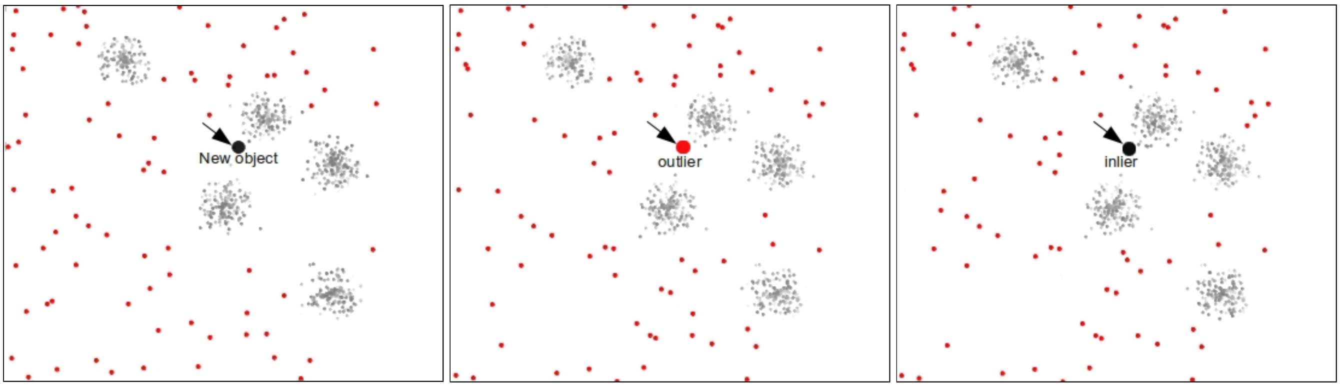
The improved efficiency of COD (Continuous Outlier Detection) stems from the adoption of an event-based approach. Instead of checking each object continuously, the algorithm computes the next time point in the future when, due to object departures, an object may become an outlier and inspects an object only at that time point. In addition, a priority queue structure is used to store objects. Such a structure supports the detection of the objects that *may* have become outliers in $O(1)$ as explained in full detail in [7]. The main difference from STORM is that the number of objects needed to be examined in each slide is significantly pruned. Also, compared to Abstract-C, it is faster and requires less space. Moreover, in [7], it is also shown how COD is adapted for multiquery cases, i.e., outliers according to multiple values of R and k are detected concurrently, yielding another algorithm called ACOD.

MCOD (Micro-cluster-based Continuous Outlier Detection) builds on top of COD and employs the same event queue. Its distinctive characteristic is that it mitigates the need to evaluate range queries for each new object with respect to all other active objects, which is common in proposals such as [8, 2]. The solution is based on the concept of evolving micro-clusters that correspond to regions containing inliers exclusively. Then the range queries for each new object are performed with respect to the (fewer) micro-cluster centers instead of the preceding active objects. In realistic data with few outliers and dense regions, MCOD exhibits the best performance. Both COD and MCOD have been re-engineered and incorporated in the extended MOA.

In summary, the approach in [2] has acceptable memory requirements ($O(kn)$), negligible time requirements to update the information for each existing object due to the arrival of new objects and the expiration of old objects ($O(1)$ for each new object), and significant time requirements to produce outliers ($O(nlogk)$). On the other hand, the approach in [8] has high memory requirements $O(n{*}windowsize)$, high runtime requirements to update existing information due to changes in the window population ($O(n{*}windowsize)$ per new object), and low time requirements to produce the actual outliers ($O(n)$). In addition, both approaches require a range query with regard to all current objects for each new object's arrival. COD and MCOD have $O(n)$ space requirements and they are faster than the exact algorithms of both [2] and [8].

4. DEMONSTRATION SCENARIOS

The demonstration of continuous outlier detection algorithms will be performed using MOA's visualization functionality. We focus on two different aspects of continuous outlier detection: i) the exploration of the outlier and inlier evolution and ii) the performance comparison among the



(a) arrival of a new object (b) the object becomes an outlier (c) the object becomes an inlier

Figure 4: Example of outlier evolution.

implemented algorithms. The demonstration will be based on both synthetic as well as on real-world data sets with different characteristics.

4.1 Exploring Data Evolution

This part of the demonstration involves the exploration of data streams with respect to outliers and their evolution. In this part, we will demonstrate visually the way objects change their status (from inlier to outlier and vice versa) during their lifetime inside the sliding window.

An example is illustrated in Figure 4, where in 4(a) a new object is inserted and become an inlier, in 4(b) the same object becomes an outlier due to object expiration in its R-neighborhood and finally in 4(c) the object becomes again an inlier due to the arrival of new objects in its R-neighborhood. Evidently, by setting different values for the parameters k and R the outliers change accordingly, and we are able to spot the timestamps that these changes took place.

4.2 Comparison of Algorithms

In the second part of the demonstration, we will show a performance comparison among the implemented algorithms. In particular, first we will show: i) how STORM compares to Abstract-C, ii) how Abstract-C compares to COD and MCOD and iii) how COD compares to MCOD. The MOA framework, supports pairwise comparison of algorithms, which is a very useful feature. Each time an update takes place, i.e., an arrival followed by an expiration, the time required to handle this update is monitored and online aggregate statistics are continuously displayed as also shown in Figure 3. At the end of execution a summary is displayed containing important information related to the efficiency of the compared algorithms such as the runtime, the number of range queries executed and the maximum amount of allocated memory.

5. CONCLUSIONS AND FUTURE WORK

In this demonstration, we study the problem of outlier detection over a data stream. In particular, we describe four online algorithms that currently are at the state-of-the-art in the topic. We have implemented these algorithms in the MOA framework, after having extended its functionality with outlier detection capabilities. By using the visualization functionality of MOA we were able to illustrate the output of the algorithms in an intuitive and clear way. We will offer the implemented outlier extensions to the MOA community. In addition, we plan to evaluate *density-based* outlier detection algorithms over data streams and extend MOA accordingly.

Another important issue for further investigation is the ability to visualize data streams in any metric space. Currently, only multidimensional data sets are supported (in particular, 2D data sets) due to visualization limitations. The challenge is to integrate into MOA effective visualization of general metric spaces.

6. REFERENCES

[1] C. C. Aggarwal, editor. *Data Streams - Models and Algorithms*, volume 31 of *Advances in Database Systems*. Springer, 2007.

[2] F. Angiulli and F. Fassetti. Distance-based outlier queries in data streams: the novel task and algorithms. *Data Mining and Knowledge Discovery*, 20(2):290–324, 2010.

[3] A. Bifet, G. Holmes, B. Pfahringer, P. Kranen, H. Kremer, T. Jansen, and T. Seidl. Moa: Massive online analysis, a framework for stream classification and clustering. *Journal of Machine Learning Research - Proceedings Track*, 11:44–50, 2010.

[4] F. Cao, M. Ester, W. Qian, and A. Zhou. Density-based clustering over an evolving data stream with noise. In *SDM*, 2006.

[5] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB Conference*, pages 426–435, 1997.

[6] E. Knorr and R. Ng. Algorithms for mining distance-based outliers in large data sets. In *VLDB*, 1998.

[7] M. Kontaki, A. Gounaris, A. N. Papadopoulos, K. Tsichlas, and Y. Manolopoulos. Continuous monitoring of distance-based outliers over data streams. In *ICDE*, pages 135–146, 2011.

[8] D. Yang, E. Rundensteiner, and M. Ward. Neighbor-based pattern detection for windows over streaming data. In *EDBT*, pages 529–540, 2009.