2016 IEEE 16th International Conference on Data Mining

Subspace Outlier Detection in Linear Time with Randomized Hashing

Saket Sathe
IBM T. J. Watson Research Center
Yorktown Heights, New York 10598
Email: ssathe@us.ibm.com

Charu C. Aggarwal
IBM T. J. Watson Research Center
Yorktown Heights, New York 10598
Email: charu@us.ibm.com

***Abstract*—Outlier detection algorithms are often computationally intensive because of their need to score each point in the data. Even simple distance-based algorithms have quadratic complexity. High-dimensional outlier detection algorithms such as subspace methods are often even more computationally intensive because of their need to explore different subspaces of the data. In this paper, we propose an exceedingly simple subspace outlier detection algorithm, which can be implemented in a few lines of code, and whose complexity is *linear* in the size of the data set and the space requirement is *constant*. We show that this outlier detection algorithm is much faster than both conventional and high-dimensional algorithms and also provides more accurate results. The approach uses randomized hashing to score data points and has a neat subspace interpretation. Furthermore, the approach can be easily generalized to data streams. We present experimental results showing the effectiveness of the approach over other state-of-the-art methods.**

I. Introduction

An outlier is a data point that deviates sufficiently from other points to give rise to the suspicion that it was generated by another mechanism. Outlier detection methods have been used in a variety of application domains such as intrusion detection, financial fraud, medical diagnosis, law enforcement, and earth science. The most popular algorithms for outlier detection include the use of distance-based methods [7], [11], [15], [21], [23]; in spite of their antiquity, these algorithms have endured as the most popular algorithms and provide surprisingly robust results. Detailed discussions on various outlier detection algorithms may be found in [1].

A particularly difficult case of outlier detection is the high-dimensional case [2] in which irrelevant attributes hide outliers. A pictorial example is used in [2] to show how irrelevant attributes have a masking effect on the outliers. Therefore, it is often helpful to explore different subsets of dimensions in which the outliers are discovered. Subsequently, a number of different methods such as feature bagging [17], high-contrast methods [14], statistical subspace selection [19], [20], and spectral methods [13], [24] have been used to score points as outliers in the high-dimensional case. In spite of these advances, both the complexity and the robustness of these methods often turns out to be a challenge. For example, *each trial* of the ensemble method in [17] requires $O(n^2)$ time for a data set containing n points because the LOF method is used for each trial. The techniques in [14], [19], [20] require expensive subspace selection operations, and in some cases [14] require multiple executions of an $O(n^2)$ detector. The spectral methods in [13], [24] require the eigendecomposition of an $O(n^2)$ matrix, which has a complexity of $O(n^2 \cdot l)$ time and $O(n^2)$ space, where l is the number of extracted eigenvectors. This type of complexity is significant because it restricts the use of these algorithms to only small data sets. For example, even a data set containing 10,000 points often requires significant time with an $O(n^2)$ algorithm.

In this paper, we propose an *extremely simple* and *fast* outlier detector that relies on randomized hashing. The complexity of the detector scales linearly with the size of the data set, and the constant factor seems to be relatively small. In our benchmarking results, we show that the detector is *one order of magnitude* faster than most of the competing detectors and is sometimes more than two orders of magnitude faster than high-dimensional methods. The outlier detector requires only a few lines of code to implement, needs constant space, and is extremely accurate. It is easy to generalize this detector to the case of data streams. Furthermore, the results from the detector are interpretable and provide a good description [5] of the outliers.

This paper is organized as follows. Our approach for outlier detection is introduced in section II. The extension of the approach to data streams is discussed in section III. Experimental results are presented in section IV. Related work is discussed in section V. The conclusions are discussed in section VI.

II. The Subspace Hashing Approach

Before describing the subspace hashing approach in detail, we will introduce some notations and definitions. It is assumed that we have a data set D, in which the ith row corresponds to the ith data point. The data set contains n points and d dimensions. The ith row is a d-dimensional vector denoted by $\overline{X_i} = (x_{i1} \ldots x_{id})$.

The subspace hashing approach works by creating a hashed representation of the data and is inherently defined as an ensemble-centric approach, in which each individual detector is *extremely* weak, but the overall performance is extraordinarily good. The approach works with a data sample of size s in order to create a training model. Subsequently, all data points (including those in the training sample) are scored with respect to this training model. The data sample is used to maintain counts of the number of training points in different

2374-8486/16 $31.00 © 2016 IEEE
DOI 10.1109/ICDM.2016.62

bounding regions in localized subspaces. To maintain these counts efficiently, the approach uses the count-min sketch [12], in which w pair-wise independent hash tables are used.

As discussed earlier, the approach uses m different executions of the base subspace outlier detector. The scores of each point over the m different components are averaged. The approach is referred to as *RS-Hash*, corresponding to the fact that it is a Randomized Subspace Hashing algorithm. In the following, we describe the steps used in each execution of a *single* component of the ensemble-centric *RS-Hash* method:

1) Select a value of the *locality parameter* f uniformly at random from the range $(1/\sqrt{s}, 1-1/\sqrt{s})$.
2) Generate a d-dimensional random vector $(\alpha_1 \ldots \alpha_d)$, such that each α_i is drawn uniformly at random from $(0, f)$. The parameter α_i is referred to as the *shift parameter*.
3) Select an integer value of r uniformly at random between $1+0.5 \cdot [\log_{\max\{2,1/f\}}(s)]$ and $\log_{\max\{2,1/f\}}(s)$. Select r dimensions from the data set and denote this set of dimensions by V. Values of r larger than d are set to d.
4) Randomly sample the data set D for a training sample S of s points. Determine the minimum value min_j and maximum value max_j of each dimension j in sample S, and normalize each $\overline{X_i} = (x_{i1} \ldots x_{id})$ in the sample as follows:
$$x'_{ij} \Leftarrow \frac{x_{ij} - min_j}{max_j - min_j} \quad (1)$$
For each of the s normalized points $\overline{X'_i} = (x'_{i1} \ldots x'_{id})$, create a new vector $\overline{Y}_i$ and set y_{ij} to -1, if that dimension is not included in V. Otherwise, set y_{ij} to the integer value of $\lfloor (x'_{ij} + \alpha_j)/f \rfloor$. Apply w different hash functions $h_1(\overline{Y}_i) \ldots h_w(\overline{Y}_i)$ to $\overline{Y}_i$ using the implementation described in section II-A. Increment the $h_k(\overline{Y}_i)$th element of the kth hash table by 1.
5) **Scoring step:** Transform each point $\overline{X} \in D$ to $\overline{Y}$ using the same steps as used for transforming points in S in step 4. The values of min_j and max_j computed on sample S in step 4 must be used. Apply hash functions $h_1(\overline{Y}) \ldots h_w(\overline{Y})$ to the transformed point $\overline{Y}$. Let the value of the $h_k(\overline{Y})$th cell in the kth hash table be c_k (zero if the cell is empty). If the point $\overline{Y}$ was originally included in the training sample then report $\log_2(\min\{c_1 \ldots c_w\})$ as the outlier score; otherwise, report $\log_2(\min\{c_1 \ldots c_w\} + 1)$ as the outlier score.

Lower values of the score are more indicative of a greater degree of outlierness. The adjustment of the score in the last step, depending on whether or not the point is included in the training sample, is needed to avoid overfitting and differential treatment of the training points. Note that in-sample points will always map to a hash table entry of at least 1, and therefore, their minimum score will always be $\log_2(1) = 0$ just like the out-of-sample points.

This process is repeated multiple times over m samples, and the scores are averaged to provide the final result. In other words, if $Score^j(i)$ is the score of the ith point over the jth ensemble component, then the final score $RSHash(i)$ is defined as follows:
$$RSHash(i) = \frac{\sum_{j=1}^{m} Score^j(i)}{m} \quad (2)$$
Each execution of the base subspace outlier detector is completely independent of the previous ones, and we always start off with the unmodified data set. Note that the use of the logarithm in the final scoring step is important and it has a significant effect on the final result. As we will discuss in section II-B, the use of the logarithm in the scoring function is related to the notion of a log-likelihood estimate. It is also assumed that the intermediate data structures such as the hash tables are always initialized at the beginning of the execution of each base component.

The sample size s is typically a small constant number of points. The basic idea here is that the level of approximation of the subspace model *in a single ensemble component* depends on the absolute sample size rather than that of the base data. The advantage of the larger base data is obtained by averaging over more independent ensemble components. Furthermore, since the scores over different executions are averaged, the approximate results over each ensemble component are significantly sharpened [4]. In fact, the sampling process adds to the diversity of the approach. We found that it was sufficient to set s to 1000 points. However, if the size of the data set was less than 1000, then the entire data set was used. In other words, the value of s was set to $\min\{1000, n\}$.

Implementation Details: An important implementation detail is that when $max_j = min_j$ for some sampled dimension j in V, such a dimension is dropped from V in step 4. Furthermore, we can drop the dimensions not included in V from the discrete representation $\overline{Y}_i$ for better efficiency, instead of setting those values to -1. Note that the normalization (Equation 1), integer conversion and hashing of a data point in step 4 should be done in one shot for best efficiency. Furthermore, if enough memory is available, it makes sense to build the hash tables of all ensemble components simultaneously, and scan the test data only once in the scoring step. This saves on computational time at the expense of space. In fact, simultaneous construction of ensemble components is essential in the streaming setting because of the one-pass requirement. Although we do not investigate the parallel setting in this paper, the approach is embarrassingly parallel because of the independence of ensemble components.

A. Implementing the Hash Function

There are two different ways in which the hash function can be implemented. The first method [12] can be used for data streams, and is extremely fast but it allows collisions, and therefore it can sometimes make mistakes. The second (simplified) method, which is similar to a closed hash table with linear probing, should be used in all static settings. The second method cannot, however, be used in streaming settings; it can only be used in settings in which we know the maximum number of entries in the table in advance.

1) The first method *RS-Hash(S)* (i.e., the sketch variant) is based on the idea of count-min sketches. The hash function $h_k(\overline{Y})$ maps each point $\overline{Y}$ to an integer value in the range $(0, p-1)$, where p is the number of elements in the hash table. We also refer to p as the *hash range*. The value of p should be selected based on memory availability in the streaming setting, although the requirements are far more modest in the static setting. After the hash functions have been computed, the value of $h_k(\overline{Y})$th element in the kth hash table is incremented by 1, as discussed in the previous section. It is noteworthy that one can use smaller values of p, if memory is a serious constraint. This is the reason that the approach can even be used in data streams where there is no bound on the number of entries that can be inserted into the hash table. Typical values of p and w are $p = 10,000$ and $w = 4$, which leads to a space requirement of less than 200KB for the sketch structure. Because of this property, it can even be used on specialized hardware like streaming outlier detection in in-network sensor nodes, where the constraints on memory are very severe.
2) The second method *RS-Hash(E)* (i.e., the exact variant) maintains a single closed hash table and it cannot be used for data streams. However, it is the recommended approach in the static setting. Since at most s entries are inserted in any ensemble component, one can use any standard hash table (supporting linear probing) with size p larger than s. The value of p should be set to at least ten times the maximum sample size of s in order to minimize the time spent in linear probing. Since the maximum sample size is only 1000, this is a very modest (constant space) requirement. However, in the streaming setting, it is impossible to use such an approach.

Although the first method allows collisions, the error of the approach is extremely low. We will provide a theoretical explanation for this in the next section.

B. Why does the Approach Work?

The approach is essentially a sampling method that estimates the (approximate) log-likelihood density of localized rectangular regions in the subspaces associated with a point, and then averages these density values over different localized subspaces *of different sizes*. In other words, the localized bounding boxes evaluated in different components are inherently of different sizes, and this is a key factor underlying the accuracy of the approach. As we will discuss later, the *shift* parameters further help in exploring bounding boxes in different regions of the data. Although errors are caused by collisions in the count-min sketch, the effect of these errors is smoothed out by using multiple ensemble components.

The step of selecting a subspace V of dimensionality r essentially identifies a localized subspace region for each point. The boundaries of this localized region are defined by the computation $q = \lfloor (x'_{ij} + \alpha_j)/f \rfloor$. Because the value of f is selected randomly, it will automatically test a bounding box of different size in each execution of the base detector. This automatically helps in testing local subspace regions of different sizes for each point. Note that all points for which the jth dimension lies between $f \cdot q - \alpha_j$ and $f \cdot (q+1) - \alpha_j$ will be mapped to the same value of the jth dimension. Two points are mapped to the same hash cell entry if *all* of the dimensions in V are mapped to the same range. However, there might be collisions caused by the contributions of other subspaces mapping to the same cell.

It is noteworthy that the inclusion of the random vector (or shift parameters) $(\alpha_1 \ldots \alpha_d)$ in the computation of the range is crucial for making this approach work. The inclusion of the random shifts ensures that even when the same dimension is sampled in a different ensemble component, the boundaries tested for a given point are different. This provides more diversity and tests different subspace localities of the data point over different components.

Note that if $h_1(\overline{Y}) \ldots h_w(\overline{Y})$ are the relevant counts of $\overline{Y}$ in the various hash tables, then the value of each $c_k = h_k(\overline{Y})$ is an overestimate on the count. Therefore, the minimum $\min\{c_1 \ldots c_w\}$ is still an overestimate on the count and the local density $\rho(\overline{Y})$ of that point can be estimated as:

$$\rho(\overline{Y}) \approx \frac{1 + \min\{c_1 \ldots c_w\}}{1+n} \tag{3}$$

Note that we have added a value of 1 up front to the counts (as Laplacian smoothing) to incorporate the prior belief that the point is an inlier and all points lie in its subspace locality. The corresponding log-likelihood fit is given by:

$$\text{Log-Fit}(\overline{Y}) = \log(\rho(\overline{Y})) = \log\left(\frac{1 + \min\{c_1 \ldots c_w\}}{1+n}\right) \tag{4}$$

The overall ensemble score of a given point is computed by the average log-likelihood fits over the different base components. Therefore, we have:

$$\text{Score}(\overline{Y}) = \text{AVG}_{[\text{All } V]} \log\left(\frac{1 + \min\{c_1 \ldots c_w\}}{1+n}\right)$$

In the equation above, the counts $c_1 \ldots c_w$ vary over the different components, whereas the term $(1+n)$ in the denominator remains constant. This term can be pulled out in an additive way because of the use of the logarithm. Therefore, we have:

$$\text{Score}(\overline{Y}) = \text{Const.} + \text{AVG}_{[\text{All } V]} \log\left(1 + \min\{c_1 \ldots c_w\}\right)$$

It is easy to see that this equation is the same as Equation 2 except for the additive constant. The constant portion of the summation, which depends only on the number of points in the data, is dropped from the score, as it is not germane to the *relative* values of the different scores. Therefore, the score computation of Equation 2 is fully explained by the notion of log-likelihood fits.

A natural question to ask is whether the level of approximation in the sketch-based approach causes significant degradation. Therefore, we provide an expression for this error and also substantiate this experimentally in a later section.

Theorem 1: Consider a sketch table with w hash functions and p as the range of each hash function. Let s be the sample

size in each ensemble component. Then the probability P that the approach provides exactly the same score for a test instance $\overline{X}$ as an approach that maintains exact counts in a given ensemble component is given by:

$$P \geq 1 - [1 - (1 - 1/p)^s]^w \tag{5}$$

Proof: The approach provides exact counts when no collisions occur with respect to $\overline{X}$ in at least one of the hash functions (in a given ensemble component). For a given test instance $\overline{X}$, the probability that any of the s samples collides with it is given by $1/p$. Therefore, the probability of no collisions over all samples for a particular hash function is given by $(1-1/p)^s$. The probability that no collisions occur in at least one hash function is given by $1-[1-(1-1/p)^s]^w$. ■

In order to get a feel for the accuracy of this approach, let us substitute a few typical numbers from our experimental setting. The value of s was 1000 in our experiments, whereas the value of p was 10000, and that of w was 4. First, we evaluate $(1 - 1/p)^s$, which is $(1 - 10^{-4})^{1000} \approx e^{-0.1}$. Therefore, we have:

$$P \geq 1 - (1 - e^{-0.1})^4 = 0.9999 \tag{6}$$

In other words, a single base detector provides the correct score with probability 99.99%. Furthermore, if we used $m = 100$ ensemble components, we can show that the sketch-based approach provides the correct score over all 100 components with at least 99% probability. Note that one can increase this probability to 99.99% by increasing the tiny 200KB memory requirement by another 50% to incorporate two more hash functions. For all practical purposes, exact scores are maintained by this approach. Even when the scores were not exact, the errors were averaged over many ensemble components and therefore the *relative* values of the scores are usually not affected. Note that the performance metrics of problems like outlier detection depend on the relative values of the scores rather than absolute values.

Logic of Parameter Choices: The algorithmic description in the previous section sets the values of the parameters r and f in a particular way. There is a certain logic to these choices. First, by always choosing f between $1/\sqrt{s}$ and $(1 - 1/\sqrt{s})$, we ensure that the dimensionality r of the subspace explored is at least 2. This is because the value of r is at least $1 + 0.5\log_{\max\{2,1/f\}}(s) \geq 1 + 0.5\log_{\sqrt{s}}(s) = 2$. The dimensionality selected is such that the expected number of points in the local region corresponding to each test point varies between 1 and $\sqrt{s}$. This ensures that the local regions are neither too small nor too large, and they vary enough over different ensemble components to accurately estimate the local density over different types of distributions.

C. Time and Space Complexity Analysis

The running time is divided into a training phase and a scoring (testing) phase. The training phase requires w operations for hashing each of the s training samples, where w is the number of hash functions. Since w is a small constant such as 4, it can be assumed that the training phase of each ensemble component requires $O(s)$ time. Note that $s \approx 1000$ is typically constant as well. Similarly, the testing phase requires constant time for hashing each of the data points. Since each of the n points is a test point (whether it is included in the training sample or not), the overall time for the testing phase is $O(n)$. Therefore, the entire process is *linear* in the number of data points, *and the constant factors involved are extremely small*. Even though the approach is run multiple times, our experimental results will show that the running times are often *significantly* faster as compared to competing methods *in addition to* being more accurate.

The space complexity of the approach is also constant. This is because the space taken by the hash table is $p \cdot w$, where p is the range of the hash function and w is a small constant such as 4. The value of p is typically a constant value such as 10,000, although it should typically be selected as a small multiplicative factor of s.

D. Interpretability of Discovered Outliers

The approach provides a high level of interpretability to the discovered outliers. For each predicted outlier, we examine the subspaces in which the score is particularly small. This provides a sparse bounding region within a subset of attributes. The bounding region is naturally defined by the computation $q = \lfloor (x'_{ij} + \alpha_j)/f \rfloor$, which maps all the data values x_{ij} within a particular range to the same value. For example, if the jth dimension is the *Age* attribute, then all points for which the *Age* lies between $f \cdot q - \alpha_j$ and $f \cdot (q+1) - \alpha_j$ will be mapped to the same value of the jth dimension. Two points are mapped to the same hash cell entry if *all* of the dimensions in V are mapped to the same range. This provides natural insights as to *why* a data point should be considered an outlier.

III. Extension to Data Streams

Because of its straightforward hashing approach, the methodology can be easily extended to data streams. In the streaming setting, the patterns in the data may change over time, and the discovered outliers should be sensitive to the changes in these patterns. There are two common ways of discounting past history:

1) One can compute the outlier scores based on only a sliding window of data points.
2) One can compute the log-likelihood density model using time-decayed scores.

It is almost trivial to implement the sliding-window approach because the count-min sketch allows both the insertion and deletion of items. Incoming points are added to the sketch, and the points falling off from the trailing end of the sliding window are removed from the sketch. The performance of the approach is also similar to the static case.

Therefore, we will focus on the more difficult setting of time-decayed scores. Time-decayed scores are sometimes more desirable because of their smooth discounting of the underlying data points. Most of the known streaming methods do not work in this time-decayed setting because of the

difficulty of maintaining the statistics of score computation in time-decayed fashion. However, the hash-based technique is able to achieve this goal with relative ease because of its use of frequency-based scores, which can easily be constructed in a time-decayed fashion.

First, we define the notion of a time-decayed score based on a *decay rate* λ. The decay rate defines the rate at which the weight of each point is reduced after the arrival of an additional data point. Specifically, the weight of each point is $2^{-\lambda t}$ after t points have arrived in the data stream. Therefore, the half-life t_h of each point is $1/\lambda$, and is intuitively equal to the *number of points* that arrive after which the weight of the point drops by a factor of 2. One key difference in the implementation of the streaming setting and the static setting is that the *counts for all ensemble components* need to be maintained simultaneously in the same sketch structure. In this case we also need to prepend the ensemble component identifier to the discrete representation that is hashed.

A few parameters are first set up in a preprocessing phase:

1) The values of min_j and max_j, which are the minimum and maximum ranges of the jth dimension, are estimated over an initial sample of the data. Although it is recognized that the minimum and maximum values might change with more incoming points, only rough estimates are required for the approach to work.
2) The *effective* number of points in each ensemble component depends on the decay rate. Specifically, the effective *weight* of all the data points is given by the geometric series $\sum_i 2^{-\lambda i} = 1/(1-2^{-\lambda})$. Therefore, we assume that the *effective* sample size s is $\max\{1000, 1/(1-2^{-\lambda})\}$. This effective sample size is only a theoretical construct that we need for selecting subsequent parameters and is otherwise not used.
3) If there are a total of m ensemble components, we sample m different locality parameters $f_1 \ldots f_m$ *up front* from the range $(1/\sqrt{s}, 1-1/\sqrt{s})$. Note that all locality parameters are selected up front because the counts for all ensemble components need to be maintained simultaneously.
4) For each $r \in \{1 \ldots m\}$ the subset of dimensions V_r is sampled up front. The number of dimensions is selected in the same way as the static case, except that the theoretical value of s from step 2 above is used in the formula for selecting the number of dimensions.
5) For each $r \in \{1 \ldots m\}$ and $j \in V_r$, the *shift* parameter α_{rj} is sampled uniformly at random from $(0, f_r)$ and stored.

Once these parameters have been set up, an online approach is used at the arrival of each data point with these stored parameters. The main problem with the time-decayed approach is that all counts decay by a factor of 2 at the arrival of each point. If we explicitly maintain time-decayed counts, the approach will be too slow. Therefore, we use a lazy approach in which the decay portion is updated only as needed. The basic idea is to have a hash table in which one additional piece of information is maintained in addition to the counts. The last time-stamp (i.e., arrival index of data point) at which that sketch cell was updated is maintained. Let t_l be that value and t_c be the current time-stamp. We make the following two changes to the access and update steps of the sketch table:

1) For accessing the counts, we multiply the entry with $2^{-\lambda(t_c-t_l)}$ and report it.
2) For updates, we multiply the current count entry in the sketch table by $2^{-\lambda(t_c-t_l)}$ and then add 1. Furthermore, the time-stamp of that entry is updated to t_c.

The online portion of the approach proceeds as follows. When a data point $\overline{X_i} = (x_{i1} \ldots x_{id})$ arrives, we normalize it using Equation 1 and the values of min_j/max_j computed during the preprocessing phase. We first compute its outlier score using the current state of the hash table (testing step) and then update the counts in the underlying entries (training update). Therefore, the training and testing steps proceed simultaneously as follows:

1) For each $r \in \{1 \ldots m\}$ and $j \in V_r$ compute $y_{ij}^r = \lfloor (x_{ij} + \alpha_{rj})/f_r \rfloor$. Set all other y_{ij}^r to -1. Let the resulting data points be $\overline{Y_i^1} \ldots \overline{Y_i^r}$.
2) For each $r \in \{1 \ldots m\}$ and $k \in \{1 \ldots w\}$ compute $h_k(\overline{Y_i^r})$ and then adjust the counts of each of the entries by using the decay-centric approach above. Let the decay-adjusted counts be c_k^r for the kth hash table and rth ensemble component. Compute the score for each ensemble component r as the $\log(1 + \min\{c_1^r \ldots c_w^r\})$. These scores are averaged over $r \in \{1 \ldots m\}$ to provide the score of the data point.
3) Update the counts in each of the w entries and m ensemble components using the decay-centric update above. Update the time-stamps of these entries as the current time-stamp.

The approach continuously reports outlier scores of new data points as they arrive, which are then used to update the model. The approach is referred to as *RS-Stream*. Note that a small constant number of operations are required for each data point, and therefore the approach is extremely efficient. This is particularly convenient in the streaming setting.

IV. Experimental Results

In this section, we will extensively compare the accuracy and efficiency of *RS-Hash(E)* and *RS-Hash(S)* with state-of-the-art outlier detectors on both static and streaming data sets. We will first introduce the data sets and performance metrics. Then, we will present the experiments for static data sets, followed by streaming data.

A. Data sets

All data sets in this paper were obtained from the UCI Machine Learning Repository[1] after some preprocessing to make them suitable for outlier detection. A summary of the data sets is shown in Table I. Many of these data sets contain more than one class label and the data set has variable overall

[1] http://archive.ics.uci.edu/ml/datasets.html

class distribution. We created outlier detection data sets by following some commonly used principles in the literature. If the data set already contained rare classes, they were included as outliers and the remaining points were inliers. When the classes were almost balanced, we significantly downsampled one of the classes and included it as outliers, while the non-sampled classes were included as inliers. Overall, we created 8 static data sets and 2 streaming data sets.

Table I
SUMMARY OF THE DATA SETS.

Data set	Points	Attribute	Percent Outliers (%)
STATIC DATA SETS			
LYMPHOGRAPHY	148	18	3.4
ECOLI	336	7	2.7
YEAST	1,364	8	4.8
CARDIO	1,831	21	9.6
MUSK	3,062	166	3.1
WAVEFORM	3,509	21	4.7
OPTDIGITS	5,216	64	2.9
KDDCUP99	25,000	41	0.7
STREAMING DATA SETS			
ACTIVITY	21,383	51	10.0
KDDCUP99-T	25,000	41	0.7

Specifically, for the LYMPHOGRAPHY data set, classes 1 and 4 were outliers while the others were inliers. The ECOLI data set contained 8 classes, out of which we used classes *omL, imL* and *imS* as outliers and the rest were included as inliers. The YEAST data set contained 10 classes. The classes *ME3*, *MIT*, *NUC* and *CYT* were included as inliers, while 5% of the total number of inliers were replaced by randomly sampling points from the remaining classes and are included as outliers. The CARDIO data set contained measurements of fetal heart rate signals. The classes in the data set were *normal, suspect,* and *pathologic*. We discarded the *suspect* class. The normal class was marked as *inliers*, while the *pathologic* class formed the outliers. The MUSK data set contained several descriptors of musk and non-musk molecules. Non-musk classes j146, j147, and 252 were combined to form the inliers, while the musk classes 213 and 211 were added as outliers without down-sampling. The WAVEFORM data set contained three classes (namely 0,1, and 2) of waves. We sampled 10% of the points from class 0 and included them as outliers, while all instances of the other classes were included as inliers. OPTDIGITS was a data set that contained digits in the range of 0-9. The instances of digits 1-9 were inliers, whereas instances of digit 0 were down-sampled to 150 points and treated as outliers. The KDDCUP99 data set was a network intrusion data set from the KDD Cup Challenge, 1999. We took a contiguous subset of 25,000 data points and marked all data points corresponding to intrusion attacks as outliers while excluding the DDoS (Denial-of-Service) attacks from the data set because of their copious nature. The other packets were marked as inliers.

Next, we describe the streaming data sets. The ACTIVITY[2] data set was a temporally ordered data set, which described several subjects performing different activities (walking, running, nordic walking, etc.) and several parameters were measured using body-mounted sensors. Thus the data is a multidimensional time series. For constructing an outlier detection data set, we took a subject's *walking* data and replaced 10% of the instances with *nordic walking* data, which were designated as outliers. The KDDCUP99-T (streaming version) was the same data set as KDDCUP99, with the only difference that temporal ordering information was included. Including temporal ordering information was necessary for streaming outlier detection.

B. Performance Metrics

For measuring performance, the standard method used for evaluating (score-wise) outlier detectors is the area under the curve (AUC) of the Receiver Operating Characteristics (ROC) curve. This metric is described in detail in [1]. The same metric was used for the static and streaming data sets. For measuring efficiency, we computed the total time in seconds for each detector in the static case. In the streaming case, we measure the performance of each detector in terms of the number of tuples processed per second.

C. Experiments on Static Data

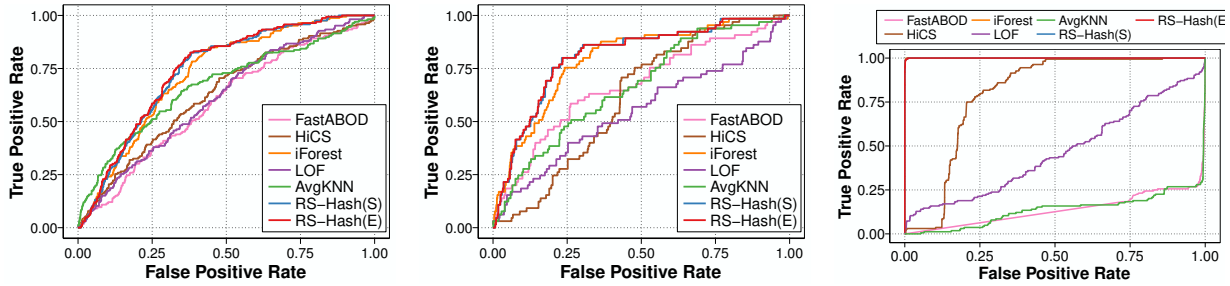
In this section we will discuss the accuracy and efficiency experiments on the static data sets. As baselines we used several outlier detection techniques such as LOF [11], AvgKNN [7]), FastABOD [16], iForest [18] and HiCS [14]. Note that three of these techniques [14], [16], [18] are specifically designed for high-dimensional outlier detection and are intentionally chosen to be different types of methods. We compared the baselines with the proposed exact method *RS-Hash(E)* and sketch-based method *RS-Hash(S)*. The parameter denoting the number of k-nearest neighbors is used in many techniques (LOF, AvgKNN, HiCS, and FastABOD). For consistency, we set its value as $k = 10$. The number of components is set as $m = 300$ for *RS-Hash(E)*, *RS-Hash(S)*, and iForest. *RS-Hash(S)* used $w = 4$ hash tables of range $p = 10,000$. The remaining parameters were set as follows. In HiCS, the number of Monte Carlo trials was set to 100, $\alpha = 0.1$, and candidate cutoff was set to 50.

1) Accuracy Analysis: We executed all the baselines and the proposed methods with the aforementioned parameter setting on the static data sets in Table I. The AUC obtained from all these runs is shown in Table II. We also show the detailed ROC curves for three of these data sets in Figure 1. It is evident that *RS-Hash(E)* and *RS-Hash(S)* consistently outperformed the remaining methods on all the data sets, although the iForest method came close in some cases. For analyzing which data sets show promising performance, we compute the improvement in AUC (over the average performance of the baselines) for each of the data sets. The largest

[2]https://archive.ics.uci.edu/ml/datasets/PAMAP2+Physical+Activity+Monitoring

Table II
COMPARISON OF AUC. THE TOP-2 AUCS ARE IN BOLDFACE.

Data set	RS-Hash(E)	RS-Hash(S)	AvgKNN	LOF	iForest	HiCS	FastABOD
LYMPHOGRAPHY	**100.0**	**99.85**	98.16	97.18	99.71	86.78	52.32
ECOLI	**88.41**	**88.44**	87.62	86.29	85.31	73.72	84.66
CARDIO	**91.61**	**91.78**	70.47	59.67	91.52	53.04	58.74
MUSK	**100.0**	**100.0**	24.48	39.99	**100.0**	47.77	23.95
OPTDIGITS	**76.04**	**76.14**	39.59	61.54	72.66	39.05	72.07
YEAST	**80.87**	**80.80**	66.47	55.06	79.44	61.23	66.91
WAVEFORM	**74.42**	**73.85**	66.98	61.08	72.47	62.61	58.91
KDDCUP99	**99.97**	**99.98**	13.6	46.43	**99.98**	79.94	13.71



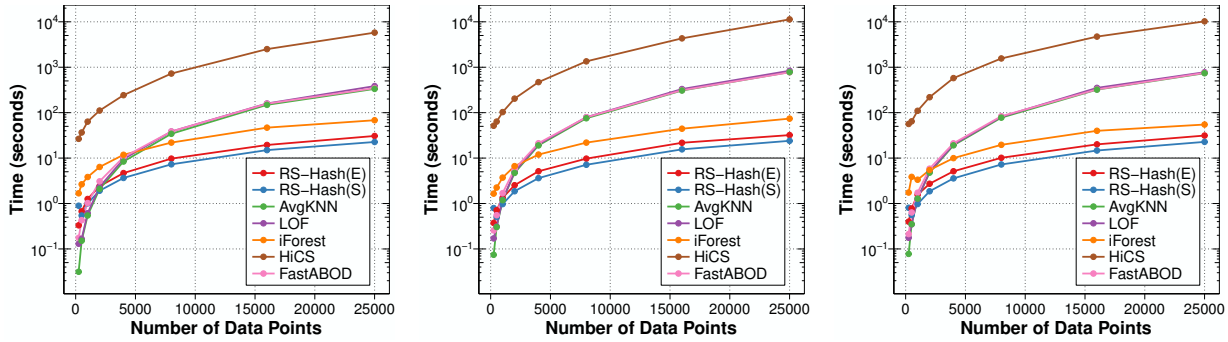
(a) WAVEFORM (b) YEAST (c) KDDCUP99

Figure 1. [Best viewed in color] Receiver operating characteristics (ROC) Curves showing the performance of all the methods. Notice that the difference between the ROC curves of *RS-Hash(E)* and *RS-Hash(S)* is negligible.

average improvement of 53% is observed on the MUSK data set, while the least improvement of 4.8% is observed on the ECOLI data set. MUSK is a very high dimensional data set. This demonstrates the effectiveness of randomized feature selection strategy of the *RS-Hash* method. The *consistency* of the approach is particularly notable because many of the baselines performed extremely poorly in at least one or more data sets, whereas the hashing method did not perform poorly on any of the data sets. This is particularly desirable in unsupervised problems like outlier detection, where robustness is paramount. Another important observation is that there is virtually no difference between the sketch-based [*RS-Hash(S)*] and exact version [*RS-Hash(E)*] of the randomized hashing method, and the differences between them can be primarily attributed to small random variations. This is an issue that we will revisit in a later section.

2) Efficiency Comparison: The next set of experiments analyze the efficiency of the proposed methods *RS-Hash(E)* and *RS-Hash(S)* with increasing size of the data set. We use both real and synthetic data sets for these experiments. Note that the outlier detectors studied in this paper are generally not very sensitive to data characteristics in terms of *efficiency*. Any minor differences are not significant enough to make any real dent in the *relative* performance over different data sets.

We sampled the KDDCUP99 data set to create real data sets of varying sizes. To ascertain that this particular choice of the data set has little influence on the execution time, we used two synthetic data sets with the same dimensionality and the number of points as this data set, but with completely different distributions. The first data set (denoted by NORMAL(0,1)) contains features drawn from a standard normal distribution with zero mean and unit variance, whereas the second data set (denoted by UNIFORM(0,1)) contains features drawn from a uniform distribution in $[0, 1]$. The results for all these data sets are shown in Figure 2. Note that most of the performance differences between various data sets can be attributed to the fact that the KDDCUP99 data contained many integer-valued attributes, whereas the two synthetic data sets contained real-valued attributes (which increased distance computation time). Since our algorithm was orders of magnitude faster than many methods, the only way we could meaningfully show these results was to use *a logarithmic scale* for the Y-axis. The differences were staggering. First, the difference between our method and competing methods almost always increased with increasing data size because our approach has linear scalability in contrast to the quadratic scalability of methods like LOF. For example, *RS-Hash(S)* is between 20 and 100 times faster than LOF for data sets containing 25,000 points. Furthermore, our approach is nearly 400 times faster than HiCS at this data size. This difference only increases with data size, and the only reason we have not shown the performance for larger data sizes is that it was not possible to execute the baselines in a reasonable amount of time. This means



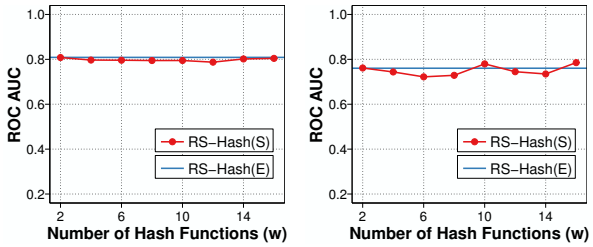
(a) KDDCUP99 (b) NORMAL(0,1) (c) UNIFORM(0,1)

Figure 2. [Best viewed in color] The execution time of all the detectors as a function of the number data points. The results are shown for one real and two synthetic data sets. **Note that the y-axis of all the figures is on a *log*-scale to enable visualization of the staggering differences.**

that the baselines are truly constrained at larger data sizes, and in these same situations the randomized hashing method is able to obtain the results in a few seconds. Therefore, these performance differences have an impact on the actual usability of our technique (versus the baseline techniques) in real settings. Notice that the choice and distribution of the data set has no influence on the efficiency ordering of these methods. *RS-Hash(E)* and *RS-Hash(S)* are always efficient by orders of magnitude than state-of-the-art distance based and subspace methods. A key driver of the superior efficiency of the *RS-Hash(E)* and *RS-Hash(S)* algorithms is its simplicity and carefully controlled linear complexity.

3) Parameter Sensitivity Analysis: In this section we will analyze the parameter sensitivity of our approach with the hash range (p) and number of hash functions (w). The memory requirements increase linearly with these two parameters. Note that the *accuracy* always improves by using larger values of these parameters, but their choice is often governed by external constraints like the amount of available memory or computational resources. Note that all experiments in this paper (including the streaming results in the next section) use a hash range of 10,000 and 4 hash functions, *which amounts to less than one megabyte of memory requirement*. Nevertheless, we show that we can obtain high accuracy with even less memory requirement than this. This means that the approach can even be used in specialized hardware architectures like in-network sensor outlier detection. Due to space constraints, all the parameter sensitivity results are only shown on YEAST and OPTDIGITS data sets. We chose these two data sets because YEAST is a smaller and low dimensional data set, while OPTDIGITS is a larger and high-dimensional data set.

The variation in AUC of *RS-Hash(S)* with the number of hash functions is shown in Figure 3. The average deviation of *RS-Hash(S)* from *RS-Hash(E)* was 1% in YEAST and 1% in OPTDIGITS. The effect of the hash range on the AUC is shown in Figure 4. Again, we observe that the accuracy of *RS-Hash(S)* is very similar to that of the exact method. This is again a consequence of Theorem 1. Concretely, the average deviation of *RS-Hash(S)* is 1.1% for YEAST and 1.3% for OPTDIGITS. The performance of *RS-Hash(E)* is shown as a straight line for reference in the same figure. Recall that in Theorem 1 we proved that for modest values of w the probability of hash collision is very low. These experiments substantiate this theoretical result by observing that the deviation of *RS-Hash(S)* from *RS-Hash(E)* is extremely small.



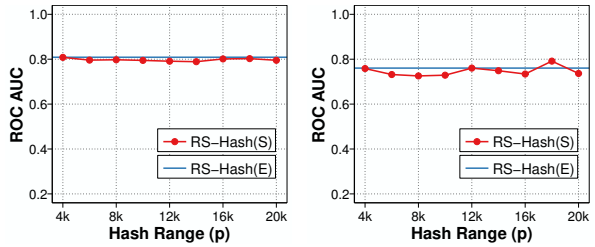
(a) YEAST (b) OPTDIGITS

Figure 3. [Best viewed in color] Sensitivity of *RS-Hash(S)* to the number of hash functions. Performance of the exact method *RS-Hash(E)* is only shown for comparison.

The performance of *RS-Hash(E)* and *RS-Hash(S)* with increasing number of ensemble components (m) is shown in Figure 5. Increasing the number of ensemble components improves both accuracy and stability. Although this paper has always used 300 components, we found that using only 100 components was more than sufficient to saturate the performance results. This means that we can gain even better efficiency than our presented results without losing much in terms of accuracy. As in other results, we found that there was little difference between the sketch-based and exact variants.

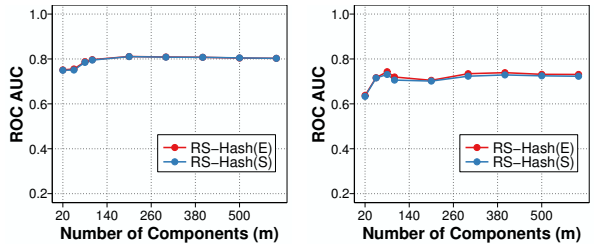
D. Experiments on Streaming Data

In this section, we show the performance results of *RS-Stream*, which is the streaming version of the algorithm. An important observation is that this approach is the first *decay-based* approach, and all previous streaming methods are window-based methods. However, we need similar decay-based baselines in order to perform a meaningful comparison. Therefore, we used decay-based adaptations of some classical



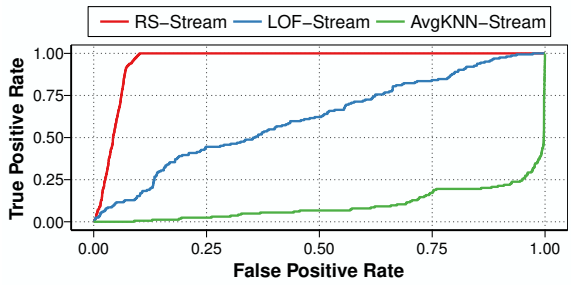
(a) YEAST (b) OPTDIGITS

Figure 4. [Best viewed in color] Sensitivity of *RS-Hash(S)* to the hash range. Performance of the exact method *RS-Hash(E)* is only shown for comparison.

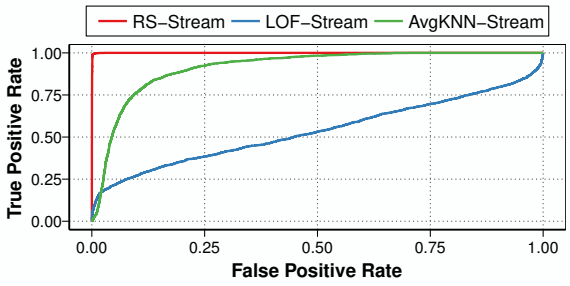


(a) YEAST (b) OPTDIGITS

Figure 5. [Best viewed in color] Sensitivity of *RS-Hash(E)* and *RS-Hash(S)* to the number of ensemble components.



(a) KDDCUP99-T



(b) ACTIVITY

Figure 6. [Best viewed in color] Receiver operating characteristics (ROC) Curves showing the performance of all the methods. The *RS-Stream* algorithm is accurate as compared to the baselines.

algorithms as baselines. In particular, we used two streaming detectors, LOF-Stream and AvgKNN-Stream, which are streaming adaptations of the LOF and the average k-nearest neighbor detector, respectively. As in the case of *RS-Stream*, we allow points to be weighted by the same decay function, except that the average k-nearest neighbor and LOF scores are computed in a weighted way with the decay function.

AvgKNN-Stream computes the score of each point as the weighted average of the distances to its k-nearest neighbors. In LOF-Stream the weights are used for computing a weighted version of the local reachability distances and the final LOF score. Another modification is that points for which the decay weight is less than the threshold of 10^{-5} are ignored. This modification also improves the efficiency of the baselines while improving accuracy in the presence of concept drift. In all experiments, the value of the decay parameter λ is set to 0.015. The number of hash tables is set to $w = 4$, hash range is set to 10000, and number of components is set to $m = 300$. The value k of the number of nearest neighbors is set to 10 for LOF-Stream and AvgKNN-Stream.

Accuracy Analysis: Table III provides a summary comparison of the performances of various streaming methods. The ROC curve for both the streaming data sets KDDCUP99-T and ACTIVITY are shown in Figure 6. Clearly, the *RS-Stream* method is highly accurate as compared to the baselines. Concretely, for the KDDCUP99-T data set the average improvement over baselines is 61% and the improvement for the ACTIVITY data set is 27%. This is because the design of the *RS-Stream* algorithm is such that the parameters of the algorithm are robust to shifts in data distributions over time. This means that *RS-Stream*'s ensemble components can survive infrequent updates. This not only makes the algorithm more efficient but also makes it more robust to changing characteristics of the data.

Table III
COMPARISON OF AUC. THE TOP-2 AUCS ARE IN BOLDFACE.

	RS-Stream	LOF-Stream	AvgKNN-Stream
KDDCUP99-T	**96.61**	**61.26**	9.7
ACTIVITY	**99.96**	53.58	**91.48**

Efficiency Analysis: The efficiency of *RS-Stream* is compared with the baselines in Figure 7. We use a metric known as *tuples per second* (TPS) to quantify the efficiency. It measures the maximum number of data points that can be processed by a particular algorithm in a given interval of time. Observe that *RS-Stream*'s average TPS over both data sets is 46 as compared to 1.5 TPS for LOF-Stream and 1.6 TPS for AvgKNN-Stream. This translates to a performance improvement of between one and two orders of magnitude, which is similar to the static case. It is noteworthy that the performance of *RS-Stream* is independent of the decay rate, whereas that of the baselines is sensitive to the decay rate because of the use of cut-off thresholds while computing nearest neighbors. In other words, if smaller decay rates were used for slowly evolving data, the baselines would perform even more slowly and would become impractical in many settings.

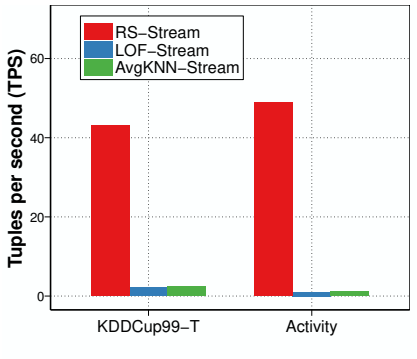


Figure 7. [Best viewed in color] Efficiency of the streaming algorithms measured as the number of tuples processed per second (TPS). Higher values are considered better.

V. Related Work

Detailed discussions and surveys on outlier detection may be found in [1]. Important classes of outlier detectors include distance-based methods [7], [10], [11], [15], [23], density-based methods [11], [21], and pattern-based compression methods [6]. Subspace outlier detection goes one step further in finding localized subsets of dimensions that emphasize the outliers [2]. Spectral methods [13], [24] have also been explored in order to discover outliers in lower-dimensional manifolds of high-dimensional data. One challenge in subspace outlier detection methods is that no single subspace is able to discover all the outliers. Recently, ensemble methods [4] have found increasing interest in the literature because of their ability to discover outliers using multiple views of the data. Starting with the work in [17], subspace outlier detection has often been explored by examining multiple axis-parallel [14], [17], [19], [20] or rotated [4] views of the data. Statistical methods for discovering relevant subspaces are explored in [14], [19], [20]. However, these methods combine feature bagging with distance-based methods, which do not seem to be well-suited to one another. Our approach is a bi-sampling technique of combining point and dimension sampling and it is better suited to the (perturbed) *subspace histogram* methodology proposed in the paper. Row-wise subsampling of data matrices have also been explored in the context of isolation forests [18] and entry-wise subsampling methods of adjacency matrices have been explored for graph outlier detection [3]. Many outlier detection algorithms have also been generalized to the streaming setting, such as distance-based methods [8], [22] and tree-based methods [9], [25], [26].

VI. Conclusions

Subspace outlier detection is an extremely challenging problem because of the high level of computational complexity associated with the identification of different subspaces. In this paper, we present an extremely simple, accurate, and linear time algorithm for subspace outlier detection with randomized hashing. Its simplicity enables implementation in a few lines of code. It has linear complexity with small constant factors and constant space requirements, which enables efficient use in very large data sets and data streams. Our experimental results show that the approach is able to achieve superior results compared to state-of-the-art methods both in terms of accuracy and efficiency.

References

[1] C. Aggarwal. Outlier Analysis, Second Edition, *Springer*, 2017.
[2] C. Aggarwal and P. Yu. Outlier detection for high-dimensional data, *ACM SIGMOD Conference*, 2001.
[3] C. Aggarwal, Y. Zhao, and P. Yu. Outlier detection in graph streams. *ICDE*, 2011.
[4] C. Aggarwal and S. Sathe. Theoretical foundations and algorithms for outlier ensembles. *ACM SIGKDD Explorations*, 2015.
[5] L. Akoglu, E. Muller, and J Vreeken. *ACM KDD Workshop on Outlier Detection and Description*, 2013.
[6] L. Akoglu, H. Tong, J. Vreeken, and C. Faloutsos. Fast and reliable anomaly detection in categorical data. *ACM CIKM Conference*, 2012.
[7] F. Angiulli, C. Pizzuti. Fast outlier detection in high dimensional spaces, *PKDD Conference*, 2002.
[8] F. Angiulli and F. Fassetti. Detecting Distance-based Outliers in Streams of Data. *ACM CIKM Conference*, 2007.
[9] I. Assent, P. Kranen, C. Beldauf, and T. Seidl. AnyOut: Anytime Outlier Detection in Streaming Data, *DASFAA Conference*, 2012.
[10] S. Bay, and M. Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. *KDD*, 2003.
[11] M. Breunig, H.-P. Kriegel, R. Ng, and J. Sander. LOF: Identifying Density-based Local Outliers, *SIGMOD*, 2000.
[12] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *LATIN*, 2004.
[13] X. Dang, B. Misenkova, I. Assent, and R. Ng. Outlier detection with space transformation and spectral analysis. *SDM Conference*, 2013.
[14] F. Keller, E. Muller, K. Bohm. HiCS: High-Contrast Subspaces for Density-based Outlier Ranking, *IEEE ICDE Conference*, 2012.
[15] E. Knorr, and R. Ng. Algorithms for Mining Distance-based Outliers in Large Datasets. *VLDB Conference*, 1998.
[16] H-P. Kriegel, M. Schubert and A. Zimek. Angle-based outlier detection in high-dimensional data. *KDD*, 2008.
[17] A. Lazarevic, and V. Kumar. Feature Bagging for Outlier Detection, *KDD*, 2005.
[18] F. T. Liu, K. M. Ting, and Z.-H. Zhou. Isolation Forest. *ICDM*, 2008.
[19] E. Muller, M. Schiffer, and T. Seidl. Statistical Selection of Relevant Subspace Projections for Outlier Ranking. *ICDE Conference*, 2011.
[20] E. Muller, I. Assent, P. Iglesias, Y. Mulle, and K. Bohm. Outlier Ranking via Subspace Analysis in Multiple Views of the Data, *ICDM*, 2012.
[21] S. Papadimitriou, H. Kitagawa, P. Gibbons, and C. Faloutsos, LOCI: Fast outlier detection using the local correlation integral, *ICDE*, 2003.
[22] D. Pokrajac, A. Lazarevic, and L. Latecki. Incremental Local Outlier Detection for Data Streams, *CIDM Conference*, 2007.
[23] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient Algorithms for Mining Outliers from Large Data Sets. *ACM SIGMOD Conference*, 2000.
[24] S. Sathe and C. Aggarwal. LODES: Local Density Meets Spectral Outlier Detection. *SDM Conference*, 2013.
[25] S. C. Tan, K. M. Ting, and T. F. Liu. Fast Anomaly Detection for Streaming Data. *IJCAI Conference*, 2011.
[26] K. Wu, K. Zhang, W. Fan, A. Edwards, and P. Yu. RS-Forest: A Rapid Density Estimator for Streaming Anomaly Detection. *ICDM*, 2014.