Detection of Structural Changes in Data Streams

Ross Callister  Mihai Lazarescu  Duc-Son Pham

Abstract

We propose new methods for detecting structural changes in data streams. Significant changes within data streams, due to their often highly dynamic nature, are the main cause in performance degradation of many algorithms. The primary difference to previous works related to change detection in data streams is our usage of an algorithmic process to define the changes. We focus on RepStream, a powerful graph-based clustering algorithm, which has been shown to perform well in a stream clustering context. RepStream, like many other algorithms, operates according to parameters which are set by the user. Primarily, RepStream uses the $K$ value to determine the degree of connectivity in its $K$ Nearest Neighbour graph structure. RepStream requires that its $K$ value be set suitably in order to achieve optimal clustering performance, which we measure in terms of F-Measure. Since real-world data streams are dynamic, with classes appearing and disappearing, and moving and shifting, this requires the $K$ value to be varied according to the current state of the stream. However, such a problem in a data stream mining context is largely unexplored. We first consider this challenge by addressing the research question: when $K$ needs to be changed. From a change detection perspective, our proposed method measures the structural variation of the underlying data stream using five different statistical and geometrical features which can be extracted whilst RepStream performs its clustering. We show that combining these features into a detection method gives promising results in regards to early detection of structural changes in data streams. We use the well known KDD Cup 1999 intrusion detection benchmark dataset, and show that our proposed method was able to identify many of the changes within the stream.

Keywords: Concept Drift, Change Detection, Stream Detection, Anomaly Detection.

1 Introduction

In this paper we propose a new method of detecting changes in a data stream by analysing features extracted from the memory contents of the graph-based stream clustering algorithm, RepStream. Data streams can vary greatly over time. This can reflect anything from a shift in customer buying habits, to anomalies in a sensor network, to attacks on a network which is being monitored (Kifer et al. 2004, Silva et al. 2013).

Given the nonstationary and complex nature of data streams, clustering them effectively has been the subject of much research recently. A major problem that this paper concentrates on is detecting when change occurs in the stream. Knowing when change occurs can be incredibly valuable information as it can be used to inform a stream clustering algorithm, for example, when to adjust its operating parameters to maintain optimal clustering, or when to drop previous data from memory in order to adjust to new patterns faster.

Our approach uses RepStream (Lihr & Lazarescu 2009) as a basis for our analysis as it is a graph-based clustering algorithm. RepStream constructs a directed $K$-nearest neighbour graph, which it used for clustering. The $K$-nearest neighbour graph structure has been used in other works previously as a method of clustering. It produces a graph in which vertexes which are close together (according to the chosen distance measure) are more likely to be connected than those which are further apart. This $K$-nearest neighbour system reflects the shape, and nature of the data, as represented in a multi-dimensional space. As such we can take advantage of this intrinsic arrangement of the data, and analyze features related to it to gain information about the dataset.

The most important operating parameter of RepStream is the number of outgoing edges each vertex has, which is often denoted as $K$. This $K$ value determines total connectivity: a higher $K$ produces more total edges in the graph and vice versa. Since RepStream uses connectivity as a fundamental part of its clustering process this results in the situation where a higher $K$ value results in fewer, more connected clusters, while a lower $K$ value results in more, less connected clusters.

In practice, it is often not trivial to determine an optimal value for $K$. This matter is further complicated by the dynamic nature of data streams. Clusters in a data stream may appear, disappear, merge, or split over time, their shapes may change, they may become more or less dense, or shift (in the sense of data points being represented by points in some $n$-dimensional space), as well as other sorts of changes. Due to the unpredictable and sometimes dramatic changes in data streams a single static $K$ value is not always guaranteed to produce optimal clustering results. Ideally, the $K$ value should be varied to match changes in the stream, as well as other possible adaptations - like increasing or decreasing the number of data points which are being stored in memory to, respectively, allow the algorithm to form a better model in the case of a stable period in the stream, or allow...
the algorithm to adapt more quickly to changes in the case of a rapidly changing stream. This has applications in the real world as it would allow an algorithm to dynamically adjust to a data stream that shifts over time.

In a practical sense our research will allow us to locate points in a data stream that correspond to changes in the underlying distribution which directly map to the performance of RepStream. This is important to our goal of determining when parameters must be varied to maintain optimal clustering performance. Having parameters set sub-optimally can dramatically affect even the most recent state-of-the-art algorithms. Our goal is to make this a non-issue, initially by determining when to change the parameters, and then in future work, to address the problem of selecting the correct values of the parameters at these change points. While this work concentrates on RepStream, the concept of extracting and examining features for use in change detection can be applied to other algorithms as well.

In this work, we first address the research question of when \( K \) needs to be varied in order for RepStream to be more optimally tuned to the structural changes in the underlying data stream. Unlike most previous work on change detection which are totally limited to the statistical nature of the data stream, our critical argument here is that the parameters of the clustering algorithm should only be varied if that brings in considerable clustering benefits. Thus, the notion of changes in this work not only means the shifts in the statistical properties of the stream, but also depends on the specific algorithm being considered. Our contributions in this work include:

- A novel perspective on structural changes that are algorithm specific, opening up future research directions for many stream clustering algorithms;
- A novel detection scheme consisting of five graph-based features and a window-based decision algorithm;
- A comprehensive analysis on the KDD Cup 1999 intrusion detection dataset.

The paper is organized as follows. Section 2 reviews related stream clustering algorithms and other change detection methods in the literature. Section 3 details our proposed method. Section 4 explains the measures used to evaluate our method. Section 5 gives a comprehensive analysis of the proposed method on the KDD Cup 1999 dataset (Stolfo et al. 2000). Finally, Section 6 concludes.

2 Related Works

There are many approaches to clustering data streams. To face the specific challenges of handling evolving data streams various approaches have been proposed. For example, by using micro clusters to record statistics about recent points, which can be clustered according to specified time-windows (Aggarwal et al. 2003, Kranen et al. 2011). Or by using subcluster structures which fade over time to make new data more relevant (Aggarwal et al. 2004, Zhou et al. 2008). Yet other approaches use grid based density structures, which decay to keep up with the new data arriving in the stream (Chen & Tu 2007).

Unfortunately, despite some novel and effective methods for handling streaming contexts, most stream clustering algorithms do not have mechanisms or methods for detecting change. Instead they often use sliding windows, where older data is discarded once it has become too old (Silva et al. 2013).

However, change in distributions of data streams over time has been a topic of research for some time. One approach maintains a reference window, and a sliding window, and compares the two using a distribution similarity function (Kifer et al. 2004). Sliding windows do ensure that newer data is used to make clustering decisions over older data, as older data may contain patterns that are no longer representative of the stream’s distribution if it changes. However, sliding windows can not solve all the challenges that a data stream can present.

Another approach to change detection in a data stream uses a minimum description length to generate code tables which ‘compress’ the distribution of a stream. When the stream is no longer optimally compressed by the code table then this marks a change in the stream (Van Leeuwen & Siebes 2008).

Yet another approach proposes a method for detecting the appearance of new classes in a data stream clustering context by deferring classification of outliers and placing them in a buffer, then analysing the points in the buffer for cohesion representing a novel class (Masud et al. 2011). Along a similar vein (Bhatnagar et al. 2014) suggests a grid based method. New points added to ExCC which do not fit into already populated cells in the grid are added to a hold queue, and when a sufficient amount of points are outside expected regions then a ‘change’ in the cluster distribution is recorded, and the grid is updated. This is an example where there is a sort of change detection in the operation of the algorithm. Unfortunately it can only detect changes related to the movement of cluster boundaries.

Other topics related to change detection in data streams include recording and tracking change over time for the identification of temporal change (Haiboo & Dunham 2011), or tracking change in a noisy stream through cluster density analysis (Nasraoui & Rujas 2006). The field of anomaly detection in data streams is also related, for example a paper by (Pham et al. 2014) which uses residual subspace analysis to detect anomalies in a compressed form of the data stream. Such anomaly detection is a form of change detection in a context where training data can be made available to determine a ‘normal’ stream state. Our approach, however, can not make assumptions about the normal state of a stream as it is likely to be unknown.

Other algorithms mention evolution in streams as a major issue (Bhatnagar et al. 2014, Forestiero et al. 2013), but as far as we are aware there are no existing algorithms that attempt to specifically locate when a data stream changes in a way that significantly affects the performance of a given clustering algorithm. Our approach differs from existing methods in that it seeks to detect arbitrary change in a data stream by using geometric features present within the \( K \)-Nearest neighbour graph structure of RepStream. That is, our approach seeks to detect locations in the stream which are likely to negatively impact the performance of clustering, and which are candidate locations for parameter adjustment.

Our approach differs from existing methods in that it seeks to detect arbitrary change by using geometric features present within the \( K \)-Nearest neighbour graph structure of RepStream.
3 Proposed Method

We propose that by extracting features from the data as it is processed by our chosen algorithm - RepStream - we can learn about when there are shifts in the underlying dataset. That is, by looking for changes in these fundamental features, we know when to expect the dataset to have shifted. By knowing when the distribution and nature of the dataset changes we can use this information to improve our clustering results.

RepStream was selected because it has been demonstrated to outperform other stream clustering approaches. Additionally, it uses both \(K\)-nearest neighbour arrangements of the data, and also the relative density of points when making its decisions. This hybrid method allows a wider range of the stream’s properties to be captured.

3.1 The RepStream Clustering Algorithm

RepStream uses a combination of graph based and density based approaches to clustering (Lühr & Lazarescu 2009). It constructs a directed \(K\)-nearest neighbour graph, adding each data point it receives from the data stream one at a time. Due to memory limitations - one can not expect an algorithm to maintain every data point from a continuous data stream in memory simultaneously - thus it uses a first-in-first-out window, maintaining only the most recent points in its \(K\) nearest neighbour graph.

The \(K\)-nearest neighbour graph is a directed graph, in which each vertex has \(K\) outgoing edges to the \(K\) nearest other vertices in the graph. Two vertices in the graph are considered to be reciprocally connected if each vertex has an outgoing edge that connects to the other vertex.

When a new data point is inserted as a vertex into RepStream’s \(K\)-nearest neighbour graph and it does not have a reciprocal connection to an already existing representative point, then that vertex becomes a new Representative point. Representative points, as their name suggests, act like representatives for other nearby vertexes. The closest Representative point that a vertex has a reciprocal connection to is the Representative point which represents that vertex. A vertex will be a member of the cluster that its Representative point is a member of.

Clustering in RepStream is done using representative points. RepStream maintains a second \(K\) nearest neighbour graph, which only its representative points are a member of. Two Representative points belong to the same cluster if they have a reciprocal connection in the representative \(K\) nearest neighbour graph, and are also density related to each other.

Density in RepStream is not absolute, it is rather the relative density that is used to determine clustering. The density relation radius of a representative point is equal to \(\alpha \times \text{AvgDist}\) where \(\text{AvgDist}\) is the average distance to its \(K\) nearest neighbours.

Figure 1: The density relation radius of a representative point is equal to \(\alpha \times \text{AvgDist}\) where \(\text{AvgDist}\) is the average distance to its \(K\) nearest neighbours.

Figure 2: An example of two representative points which are both reciprocally connected, and density related.

Figure 3: An example of two representative points which are reciprocally connected, but not density related.
3.2 Structural Changes

First, we define a stream clustering algorithm as being stable if its clustering performance (which is measured in terms of F-Measure in this work) varies little as a number of samples arriving from a stationary data stream is sufficiently large. Then, we define structural changes associated with a stable clustering algorithm as statistical or geometrical changes in the data that lead to a significant deviation in clustering outputs. In the case of RepStream, our hypothesis is that it is reflected in the structure of the data points, specifically in regards to their geometric properties within the $k$-NN graph of RepStream. We note the following:

- From the definition, it follows that structural changes may correlate strongly with distributional or statistical changes in the data stream. A data stream represents samples taken from a distribution of data over time. This distribution may change as the stream progresses, in a way such that the structure of the data also changes, for example concept drift (Tsymbal 2004). It can refer to new parts in the data distribution appearing, or disappearing, or the changing of existing parts of the distribution.

- Structural changes depend on the specific algorithm and its sensitivity against the changes in the data stream. This is an important aspect because change detection would not be useful if it does not lead to a need to adjust the underlying clustering algorithm.

3.3 Features Extraction

We have examined various intrinsic features within RepStream’s graph based clustering approach to determine whether they can be used to identify changes in the underlying data stream. The features we have concentrated on are: the cluster count over time, the number of edges created and removed over time, the number of cluster merges and splits over time, and the variation in the length of the edges over time. We note important that in order to extract these features, we need an active instance of RepStream with some value of $K$, which may not be optimal. This value of $K$ is a proxy for the detection only. This proxy $K$ value is desirably small as it leads to more efficient clustering results and be more stable than the lower $K$ values. Cluster Count, as well as being relevant to the clustering result, also correlates to the stability of the stream, as the number of clusters changes less when the stream is relatively stable.

**Edge Change Count** The number of edges created and removed at the representative level over time - the $K$-nearest neighbour change count - was chosen as a feature due to its ability to reflect the degree of change that the graph requires when data points are inserted. The idea is that when the data stream is stable then the representative points will also remain stable. Thus, the amount of changes at the representative level will be stable. On the other hand, when the data stream is shifting then it is expected that the number of edges that need to be updated at the representative level will vary, due to representatives needing to be created or destroyed. For example a graph vertex that is inserted outside existing clusters will be more likely to become a representative point, and need to cause updates in other representative points. On the contrary, a vertex inserted among a group of existing vertices can often be represented by an existing representative point, thus it does not cause any representative edge changes. This feature is extracted by counting the number of edge updates which are on representative points since the last measurement. In our experiments, it is every 500 points, or half the sliding window size.

**Cluster Merges and Splits** Counting the cluster merges and splits over time is used as a feature because it is likely to correlate with the stability of the data. RepStream creates clusters by considering the connectivity of representative points, as well as the local density of each representative point. Due to nodes being inserted, the density or connectivity of representative points may change and this results in the clusters splitting apart or merging together. When the dataset is stable, inserting new points is less likely to result in such changes. However, it may become more likely when the dataset shifts then points occurring in new locations, and being removed due to the first-in-first-out window. Based on this observation, we have selected the combined number of cluster merges and splits as a feature to examine when searching for change in the dataset. Similar to the $k$-NN change count, this feature can also be efficiently extracted by counting the number of times clusters merge and split since the last measurement.

**Cluster Merges and Splits** occur more rapidly when the structure of the stream changes due to data point distributions shifting outside established cluster boundaries. This causes the clusters to become unstable and causes splits and merges to occur until the algorithm can arrange the datapoints into a stable configuration.

**Edge Length Variation** Edge length variation is measured as follows. Each vertex in RepStream’s graph maintains outgoing connections to its $K$ nearest neighbours. The standard deviation of the length of these outgoing edges is calculated for each vertex. The standard deviations are then added together and divided by the total number of points in the $K$-NN graph to find the average standard deviation over every point in memory. The idea behind this feature is that one would expect a relatively consistent edge length variation when the dataset is stable. If new clusters were to form outside existing clusters the edge length might increase since longer edges would need to be formed to maintain the $K$-NN graph. If the density of an existing cluster were to increase then
the total edge lengths might decrease, which would similarly lead to an increase in the standard deviation in the edge lengths. This feature, therefore, is selected as a candidate for tracking changes in the dataset.

**History Count**: History count represents the number of times a point returns to a previous cluster. As the stream progresses individual data points change cluster membership. Even adding a single point can result in many points changing from one cluster to another. We keep track of the number of times each point in the first-in-first-out queue has returned to a cluster that it was previously in. Our hypothesis is that when structural changes are taking place within the dataset the clustering results will be unstable. This instability leads to a higher rate of individual points jumping between clusters over time. When the dataset is stable, on the other hand, the rate of change in cluster membership will be lower, due to new points not changing the dataset’s structure significantly.

### 3.4 Detection Scheme

We propose a simple scheme using the extracted features to determine whether a change has occurred or not. Each individual feature is examined separately using a time-series change detection algorithm. The algorithm is shown as Algorithm 1. The inputs are listed and given as \((feature, M, H, \lambda)\). \textit{Feature} represents the given feature as a time series. The parameter \(M\) is a multiplier which affects how sensitive the algorithm is to change; a higher value will cause the algorithm to require a larger shift in the time series before a change is detected. The parameter \(H\) is the number of previous points over which to track the algorithm changes which contains a list of time indexes where changes have occurred in the time series \textit{feature}.

**Data**: \((feature, M, H, \lambda)\)

**Result**: Changes: a list of indexes where changes have been detected

\[
changes = \emptyset; \\
X = \text{mean}(feature(i - M : M)); \\
\sigma = \text{standardDeviation}(feature(i - M : M)); \\
\text{for } i = M : \text{size(feature)} \text{ do} \\
\text{ \hspace{1em} } ma = \text{mean}(feature(i - M : M)); \\
\text{ \hspace{1em} } s = \text{standardDeviation}(feature(i - M : M)); \\
\text{ \hspace{1em} if } abs(ma - X) > M \times \sigma \text{ then} \\
\text{ \hspace{2em} } changes = changes + \{i\}; \\
\text{ \hspace{2em} } X = ma; \\
\text{ \hspace{2em} } \sigma = s; \\
\text{ \hspace{1em} else} \\
\text{ \hspace{2em} } X = (1 - \lambda) \times X + \lambda \times ma; \\
\text{ \hspace{2em} } \sigma = (1 - \lambda) \times X + \lambda \times s; \\
\text{ \hspace{1em} end}
\]

**Algorithm 1**: Algorithm for feature change detection

While this algorithm is written for a batch dataset rather than a stream, it can easily be modified for use in a stream since it only requires the past \(n\) data points to be stored in memory, as a sliding window.
Since there are 5 features that we are testing we use a system where multiple features must agree before a structural change is detected. At least \( N \) features must agree that there has been a change within the last \( T \) samples for the algorithm to detect a ‘change’. Where \( N \) is greater than half the number of features then it is simply majority voting.

Figure 10 shows where the changes are detected in the case of the Edge Change Count feature, where \( M = 1 \), \( H = 20 \) and \( \lambda = 0 \). Edge Change Count was selected for illustrative purposes as it gives a clear idea of what varying the parameters does. The red shows the raw value of the feature, which varies significantly from point to point, blue shows the \( X \) value over time, as well as \( X \pm \sigma \), and green shows exactly where the change points are detected. Figure 11 shows the same feature, but the \( M \) value has been changed to 30, while Figure 12 has the \( \lambda = 0.001 \), so that the \( X \) and \( \sigma \) values slowly adjust over time.

4 Algorithm Evaluation

In this paper we use an evaluation measure known as MTR - Mean Time Ratio (Bifet et al. 2013).

MTR is a combination of several important metrics, and is meant to evaluate a change detection algorithm in a single number. It is a combination of several important metrics. The formula for MTR is:

\[
MTR = \frac{MTFA}{MTD} \times (1 - MDR).
\]

Where MTFA is the mean time between false alarms, MTD is the mean time to detection, and MDR is the missed detection rate. A higher MTR is desirable, and this measure can be used to directly compare two change detection results.

MTR is chosen as our evaluation metric because of its specific design for use with change detection algorithms. Looking simply at the number of successful detections or the number of false alarms does not give a clear picture of how well an algorithm performs, because it is trivial to maximise either of those scores individually. Typically compromising between a high rate of detection, and a low rate of false alarms is desirable in practice. Mean Time Ratio takes both the false alarms and detection rate into account, and is ideal for evaluating the differences between detection results.

Also included in our evaluations are the individual values for Mean Time Between False Alarms (MTFA), Mean Time to Detection (MTD), and Missed Detection Rate (MDR). These measures indicate, respectively, the rate of false alarms, the time taken to detect changes, and the detection rate.

5 Experiments

5.1 KDD Cup 1999 Dataset

We select the well known KDD Cup 1999 intrusion detection dataset (Stolfo et al. 2000) to demonstrate the proposed method. It is made up of data extracted from a computer network being monitored during various simulated and controlled network attacks. We use the available subsampled version of the dataset which contains approximately 500,000 data points, as well as ground truth class labels for evaluation purposes.

KDD has been used previously as an example of a real-world data stream used in evaluating stream clustering algorithms (Lühr & Lazarescu 2009) (Cao et al. 2006) (Ruiz et al. 2009). The varied attacks over time simulate the dynamic and unpredictable nature of a data stream, making it ideal to test our change detection methods on. Unfortunately, however, there remain very few real world benchmark datasets available for evaluation purposes. A recent survey by (Kaur et al. 2015) indicates that a major issue in stream clustering literature is the lack of availability of benchmark datasets. As such, KDD remains perhaps the only publicly available real-world stream
Figure 13: Presence of classes during the KDD data stream. Type 1 attacks are prolonged single classes, type 2 are quick single attacks interspersed by normal, and type 3 are rapid clusters of attacks.

dataset which has the necessary traits to evaluate our methods.

**Classes in KDD** Figure 13 shows the class presence for the KDD dataset (we note that all figures are best viewed in colour). For each class in the dataset a line was plotted. The value of the line is the class label if that class is present during that time during the stream, and zero otherwise. Class 1 is the ‘normal’ traffic, and every other class represents different attacks. The spikes in Figure 13 correspond to when attacks occur, and the plateaus represent prolonged denial-of-service (DOS) attacks. The most important part of this is when attacks occur, i.e. - the spikes and beginnings and ends of the DOS attacks in the class presence plot - as this tells us when the data stream changes due to an attack occurring.

Figure 13 also shows examples of the various types of attacks. The attacks labelled with type 1 are prolonged attacks made up of only a single class. Though the actual data contained within each successive data point may vary all the instances belong to the same class. There are the relatively rare instances of type 2, which contain very short attacks from a single class interspersed with data from the normal class. Then there are type 3 attacks, which are occurrences where many types of attacks occur in rapid succession, interspersed with the normal.

**Optimal RepStream K Value** Figure 9 shows the K value which produces the highest F-Measure score over time, in blue, superimposed over the F-Measure of each K value. In the background is a grey-scale collection of cells, which we call a ‘heatmap’. At each time step (along the X axis) the K value is represented along the Y axis, and the brightness of that individual cell represents the F-Measure score at that time with that K value. That is, the brightness is between 0 (black) and 1 (white), and matches the F-Measure of that K value at that time step. The range of K values is between K = 5 and K = 30. This figure shows that the K value which produces the optimal clustering results (the brightest cell at a given time step) may vary considerably over the course of the stream. Therefore, the best performance can not be achieved by using a single K value.

Furthermore, Figure 9 also shows that at some times the range of K values which produce F-Measure results near the optimal is very large - when the bright regions are large vertically, and at other times the K value is optimal within a very specific range - when the bright regions are more constrained. This figure, when combined with the information in Figure 13, shows that when attacks occur in the data set, and when the classes shift, the optimal K value does change in response to the stream, and that to get optimal performance the stream must be monitored for such changes, and adapted to when they occur.

**Ground Truth Change Points** The class presence is what we use as the ground truth for evaluation of our technique. Whenever a class appears or disappears we define a ground truth change point. These change points are weighted according to how much they affect the clustering performance of RepStream. Where the distribution of F-Measure values changes significantly the ground truth is more heavily weighted than if the distribution changes less. The Bhattacharyya distance between the distributions is used to weight the ground truth, and any below a threshold are omitted for the sake of clarity.

The heatmap is important because it shows at which points during the data stream the RepStream algorithm performs well, and when it performs poorly. This gives an idea of when changes occur in the distribution data points over time. It is intuitive to see where major changes occur, thus it is used as as the background to give context to the results.

**5.2 Results**

Figure 1 shows a table of the results for our detection algorithm. It contains values for the Mean Time between False Alarms (MTFA), Mean Time to Detection (MTD), the Missed Detection Rate (MDR), and the Mean Time Ratio (MTR). Figure 14 shows a visual representation of the detection rate for our algorithm when optimised to give the highest possible MTR value. The parameters used were $M = 1.4$, $H = 15$ and $\lambda = 0.001$, and the MTR was 39.46, mostly due to the low rate of false alarms.

Figure 15 has the parameters optimised for a higher detection rate. The inputs were $M = 1.2$, $H = 9$ and $\lambda = 0.001$, which gives a lower MTR.
5.3 Comparison To OSVM Classification Method

We also tested the features using a different detection schema, for the sake of comparison. Using a Support Vector Machine we performed the detection as a simple two-class classification. Using a sliding we trained a One-class SVM with $H$ data points, where the data points were simply 5 dimensional vectors representing each of our 5 features at a given time step. The next $\frac{M}{2}$ data points were then classified using the OSVM, and if at least half of them were not classified into the OSVM’s trained class then it would mark a change at that point.

Figure 16 shows the results of our test using the One-class SVM detection approach. Optimising the algorithm to give the highest MTR resulted in a MTR value of 7.36, and had a much higher rate of false alarms compared to our proposed approach.

The SVM detection method results in a larger number of correctly detected changes, as evident from the lower MDR value, however this is at the expense of far more false alarms, with the relatively low value of 186.15 as the MTFA. This combination results in a lower MTR score compared to our proposed detection method.

Our method when optimised for MTR yields a significantly higher total MTR score, as well as comparatively fewer false alarms. A different optimisation for our proposed method optimises to reduce the Missed Detection Rate as much as possible while retaining a comparable MTR (Figure 15). The lower MDR of 0.20 means that 80% of changes are successfully detected, with the drawback of a higher rate of false alarms, leading to a decreased MTFA score for this method.

5.4 Results Discussion

Our algorithm detects points where the features change, as in Figures 10-12. When all features are combined into a single detection method they produce the change points, marked in green on Figures 14 and 15. Both of these methods are optimised differently.

Our method of evaluation is limited in the sense that ground truth changes only take into account when classes appear and disappear. This is a limitation of the dataset used.

Whilst our proposed algorithm performs well, it still has problems, notably in the form of false alarms and changes which are not detected. For example the false alarms at around $Time = 1.85e + 05$ and $Time = 1.92e + 05$. At these times there are no ground truth changes, as it is during a time where there is only a single class present (shown in Figure 13). However, at that time the performance of RepStream does change, so it could be explained by a change in the distribution of that single class at that time.

Despite the false alarms our algorithm detects a high amount of the ground truth changes (Table 1). Many of the changes that are not detected, too, are missed due to the close proximity of the changes. Our algorithm is limited in the case where dramatic changes happen very rapidly.

6 Conclusion

There are significant challenges in detecting changes in a dataset, particularly when the dimensionality is
high. By extracting features of the data from a K-nearest neighbour graph we reduce the problem to detecting changes in a smaller number of time series, representing structural properties of the data.

We have presented a novel method of detecting concept changes in data streams by examining structural properties of graph based arrangements of the data. This approach has been shown to work even in high dimensional data, as with KDD which was treated as a 34 dimensional dataset. We take features in RepStream’s K-nearest neighbour structure and use a time-series change detection algorithm with the goal of identifying when major changes in the underlying dataset have occurred.

Our detection algorithm outperforms a similar approach using OSVM techniques with respect to MTR on the well known KDD intrusion detection dataset. The KDD dataset was selected due to its use as a benchmark in prior literature, as well as the fact that it mirrors a real-world application of change detection.

Whilst the approach does have limitations, particularly when the changes occur rapidly, and when the changes are very subtle, it produces good results when tested on a real world dataset.

References


KDD Cup 1999 Dataset (n.d.).


Masud, M., Gao, J., Khan, L., Han, J. & Thuraisingham, B. (2011), ‘Classification and novel class detection in concept-drifting data streams under time constraints’, IEEE Transactions on Knowledge and Data Engineering 23, 859–874.


Figure 14: Our moving average detection algorithm on the KDD dataset, with parameters optimised for the highest MTR value

Figure 15: Our moving average detection algorithm on the KDD dataset, with parameters optimised for lower Missed Detection Rate

Figure 16: The OSVM comparison algorithm on the KDD dataset with parameters optimised for the highest MTR value