School of Electrical Engineering, Computing and Mathematical Sciences

# Automatically Selecting Parameters for Graph-Based Clustering

Ross Callister

This thesis is presented for the Degree of
Doctor of Philosophy
of
Curtin University of Technology

February 2020

To the best of my knowledge and belief this thesis contains no material previously published by any other person except where due acknowledgement has been made. This thesis contains no material which has been accepted for the award of any other degree or diploma in any university.

Ross Callister

# Acknowledgements

Huge thanks to my programme supervisors - Mihai Lazarescu, Duc-Son Pham, and Patrick Peursum, who provided invaluable advice and information, and who amazingly managed to somehow guide me, dazed and confused, to completing this thesis.

I would also like to acknowledge and thank the Australian Federal Government, who generously provided me with the Australian Postgraduate Award (later replaced by the Research Training Program) during my research, which ultimately allowed this work to be produced.

My deepest gratitude goes to my parents, who supported and encouraged me throughout this journey. I love you and appreciate all you've done for me. Thank you for being so amazing.

Also to my pets and friends, past and present, thank you for keeping me sane helping me smile at times where smiling was difficult.

# Abstract

Clustering data streams is an area of current and active study in the data mining community. Data streams present challenges which are unique and distinct from traditional batch datasets, namely in that each data point may only be read once, there is a potentially unlimited number of data points to cluster, and that the distribution of the data points may vary over time. This final challenge requires that the algorithm be able to adjust itself to match the changing distributions as they arise.

Because clustering is an unsupervised process, setting the input parameters correctly is of vital importance, and can greatly affect the quality of clusters produced by a clustering algorithm. However, in a data stream the distribution of the data can change over time. This means that not only is it a challenge to set initial input parameters appropriately, but the parameters used for clustering may not be appropriate at different times during the stream. In this thesis we address this issue for the state-of-the-art clustering algorithm - RepStream by removing the need to set and update the critical $K$ parameter. This leads us to introduce the RobustRepStream algorithm, an extension to its predecessor which is designed to adapt to some of the most difficult challenges facing stream clustering algorithms.

Our first major contribution is a demonstration that features computed from the structure of the $K$-nearest neighbour sparse graph used by RepStream can be used to detect change in the distribution of the dataset. We do this by computing several features based on the internal $K$-nearest neighbour graph structure and using a change detection algorithm on these features to show that they vary over time as the stream distribution changes. This in turn provides evidence that changes in the data distributions are measurable and can be used to help algorithms adapt to the changes.

Secondly, we introduce the anomalous edge score, which is a feature computed from the $K$-nearest neighbour structure of RepStream. The anomalous edge score represents the number and relative length of edges which are significantly longer than average for any given vertex. We study the average anomalous edge score for a range of $K$ values on various dataset and suggest that changes in the value of the score correspond to changes in the value of the optimum $K$ value. By introducing a threshold to the anomalous edge score we can continuously select $K$ values in RepStream over the length of the stream which perform comparably well compared to the optimal $K$ value at each time step.

Next, we modify and extend RepStream to automatically vary the $K$ parameter for newly added data points in response to the changing anomalous edge score. We adjust the $K$ parameter based on a threshold in a single live instance of RepStream. Setting the $K$ parameter in RepStream was difficult as different datasets had different optimal values, and in data streams the optimal $K$ value could vary over time as the distribution changed. We demonstrate that by adjusting the $K$ value for new incoming data points we can continuously select and update its internal $K$ value which produces high quality clustering results over time in data streams.

Finally, we propose RobustRepStream, an extension of the RepStream algorithm, to automatically select the number of outgoing edges for newly added data points in response to the changing skewness excess score. RobustRepStream causes the number of outgoing edges for each vertex to be no longer dependent on a static $K$ value. We demonstrate that this method constructs nearest neighbour graphs without the need for specifying a specific number of outgoing edges, which removes the need to set the most critical and sensitive parameter of RepStream.

# Published Work

This thesis includes material from the following works that have been published over the course of the Ph.D research. They are listed along with the corresponding chapters within this thesis.

- **Article 1 (Chapter 2,3) -** Callister, R., Lazarescu, M., & Pham, D. S. (2015). Detection of Structural Changes in Data Streams. In *Proceedings of the Thirteenth Australasian Data Mining Conference (AusDM 2015), Sydney, Australia, August 2015* (pp. 79-88).

- **Article 2 (Chapter 4) -** Callister, R., Lazarescu, M., & Pham, D. S. (2017, April). Graph-based clustering with DRepStream. In *Proceedings of the Symposium on Applied Computing* (pp. 850-857).

- **Article 3 (Chapter 2,4) -** Callister, R., Pham, D. S., & Lazarescu, M. (2019). Using distribution analysis for parameter selection in RepStream. *Mathematical Foundations of Computing*, 2(3), 215-250.

- **Article 4 (Chapter 2,5) -** Callister, R., Pham, D. S., & Lazarescu, M. (2020). RobustRepStream: Robust Stream Clustering Using Self-Controlled Connectivity Graph. *Intelligent Data Analysis*, 24(4).

# Statement of Contribution by Others

Articles in the following attribution tables are numbered according to the list of published works on the previous page. This represents contributions to the following papers by the listed co-authors.

Attribution of Dr. Duc-Son Pham

| Article | Literature Review | Concept Design | Software Modelling | Experimental Results | Data Analysis | Discussion | Paper Writing |
|---|---|---|---|---|---|---|---|
| 1 | | | | | | ✓ | |
| 2 | | | | | | ✓ | |
| 3 | | ✓ | | | | ✓ | |
| 4 | | ✓ | | | | ✓ | |

Attribution of Dr. Mihai Lazarescu

| Article | Literature Review | Concept Design | Software Modelling | Experimental Results | Data Analysis | Discussion | Paper Writing |
|---|---|---|---|---|---|---|---|
| 1 | | | | | | ✓ | |
| 2 | | | | | | ✓ | |
| 3 | | | | | | ✓ | |
| 4 | | | | | | ✓ | |

Ross Callister. ................................

Dr. Duc-Son Pham...............................

Dr. Mihai Lazarescu.............................

# Contents

# List of Figures

# Chapter 1

# Introduction and Motivation

It has been predicted that in the current age of information, the total amount of data generated and stored in all human history will double approximately every two years for the foreseeable future (Zwolenski et al., 2014). This massive explosion in data is thanks to the rising popularity and use of things such as:

- *Sensor networks* (Kumar et al., 2017) used for monitoring the state of equipment or the environment.
- *Location data* (Zhao et al., 2016) from GPS systems built into cars, smartphones, animals for behavioural research, and other devices.
- *Social networking data* (Imran et al., 2018) from online networks such as Twitter and Facebook, in which huge amounts of interactions occur every second.
- *Web metadata*, such as clickstream analysis (Wang et al., 2017) which provides analytics services for website administrators.
- *Cloud computing* (Botta et al., 2016) with the need for analysis and optimisation of usage being an ever more important thing.
- *Smart-connected devices* such as the Amazon Alexa or Google Nest which generate information about user behaviours and requests.

With the huge rise in captured data comes a proportional increase in demand for data-mining techniques, and in order to make use of this torrent of new data the field of data stream mining has arisen. Such streams present a number of constraints (Aggarwal, 2015), specifically their unbounded length, the infeasibility of storage, the one-pass constraint, concept drift, the rate of data generation, and the vast amounts of metadata which must be handled.

Data mining techniques used on data streams can include supervised classification, outlier detection, and pattern mining, however stream cluster analysis is the topic on which we focus in this work.

Cluster analysis is regarded as the primary objective of unsupervised data mining (Gorunescu, 2011), and is a vital tool in exploratory data mining. While on the surface it may seem like a trivial and intuitive task to determine which members of a group are similar and which are dissimilar, this task is actually incredibly complex in an unsupervised context, and is abound in edge cases and subjective judgements. This has led to a wide and varied field of research into determining better ways of clustering data, particularly, in recent times, on data streams.

## 1.1 Data Stream Clustering

Data streams continue to become a more common feature of study thanks to their prevalence in the computing sphere. Streams generated by sensor networks, online interaction, business or market data, and video streams contain huge amounts of data which can be mined to gain useful, practical insights (Barddal, 2019). Particularly, stream clustering is an informative process, which involves the unsupervised separation of data points into groups, called clusters.

Data streams are typified by a series of data arriving sequentially over time. This data may be numerical, categorical, binary, images, video, audio, or any one of a multitude of different types. This thesis concentrates on multi-dimensional numerical data, which can be expressed as vectors in hyperspace.

**Definition 1.1.1** *A data stream $\mathscr{S}$ is a set of vectors $X_1, X_2, \ldots$ which is unbounded in length and ordered by time:*

$$\mathscr{S} = \{X_1, X_2, \ldots, X_m \ldots\} \tag{1.1}$$

with each vector $X_i$ in the stream being composed of a number of attributes - $X_i = \{x_{i,1} \ldots x_{i,d}\}$ - where $d$ is the dimensionality of the stream, the number of attributes associated with each data point. Each attribute of a data point has a real value, such that the data point can be represented as a point in $d$-dimensional space.

The values of these attributes define the location of the vector in the hyperspace. The attributes making up each data point are also alternatively referred to as data dimensions, observations, features, or components.

Using clustering algorithms we seek to group the vectors from the data set into groups of similar data points, as distinct from data points which are dissimilar (Berkhin, 2006; Charu & Chandan, 2013). The fundamental steps of clustering (Gorunescu, 2011) are:

- Defining a similarity measure between data points.
- Defining a method to construct clusters based on the similarity measure.
- An algorithm which incrementally changes clusters as new data points arrive from the stream.

Similarity between data points is defined according to a given distance metric. Most commonly, the Euclidean distance, Manhattan distance, or Manhattan Squared distance measures are used due to their simplicity and intuitiveness (Jain et al., 1999). However, other more sophisticated distance measures are occasionally used including the Mahalanobis distance measure (Mao & Jain, 1996). While alternative distance metrics can address problems related to Euclidean and Manhattan distances, namely that attributes with greater scales tend to dominate over the smaller scaled ones, another common solution is to normalise each dimension of the incoming data points, such that the scales of each is equal.

In both stream and batch clustering methods, data points are grouped based on their distance from each other using a chosen distance metric. Thus, the process is to separate the data points $\{X_1, X_2...X_k...\}$ into subsets known as *clusters*, by mapping each point to one of a series of clusters $C_1, C_2, \ldots C_j \ldots$. These clusters are meant to approximate distinct concepts within the dataset, and describe different regions of the data. As such, clusters represent a model of the hidden underlying concepts of the stream, and this is what makes clustering an unsupervised process - that training data and validation data is unavailable to the algorithm. Clusters typically do not intersect within the data space, although exceptions exist.

Stream clustering is typically a process used on batches too large to fit entirely into memory, or unbounded streams of data, in which clusters must be produced either incrementally over time, or at user-specified times. Clusters can be used to locate sub-groups within a complex dataset, and to identify relationships between members of the stream. Cluster analysis has been used in a rage of applications such as market research, image analysis (Li et al., 2018), medical analysis (Camara et al., 2019), genome clustering (Wang et al., 2019), recommender systems (Logesh et al., 2019), and anomaly detection (Aytekin et al., 2018), and so reliable performance when dealing with datasets where little is known is important.

Due to the streaming nature of the data the membership of the clusters can change over time, with data points swapping from cluster to cluster, or being added to new clusters as the clustering algorithm dictates. Data streams have a series of requirements (Cao et al., 2006), specifically:

- The number of clusters can't be assumed.
- Clusters may be of arbitrary shape and size.
- There may be outliers and noise within the data set.

As such, clusters are typically thought of as contiguous regions of the data space which contain data points at a relatively high level of density, separated by regions of space with a significantly lower density or containing no points at all.

There are various approaches that are used to perform clustering, with one of the earliest being the well-known $K$-means algorithm (MacQueen et al., 1967), which is a partition-based approach. While the $K$-means algorithm was designed for batch datasets rather than streams, later approaches have applied the $K$-means algorithm to the streaming context (Ackermann et al., 2012). The limitation of the $K$-means algorithm is that the number of desired clusters must be specified, and the algorithm is only capable of finding convex clusters. To address these limitations, there are a wealth of other more sophisticated approaches (Aggarwal, 2015, 2007; Charu & Chandan, 2013) including micro-cluster and density based methods, graph-based methods, grid-based methods, agglomerative and divisive hierarchical approaches, and even unique evolutionary or biologically-based clustering approaches. These other approaches have their own limitations and requirements. Micro-cluster and density based approaches, for example DenStream (Cao et al., 2006), typically do not require a number of clusters to be specified, however density threshold and micro-cluster size parameters typically need to be given by the user. Graph-based approaches, such as ExCC (Bhatnagar et al., 2014) and PatchWork (Gouineau et al., 2016), similarly typically require density thresholds and grid size parameters to be set by the user. Graph-based approaches often use nearest-neighbour graphs like SNCStream (Barddal et al., 2015), or minimal spanning trees like HASTREAM (Hassani et al., 2014), which require parameters to determine connectivity, or thresholds for cluster construction after hierarchical tree splitting has occurred. As we see, whilst more sophisticated approaches can operate without specifying the number of clusters, they each have their own requirements for parametrisation.

Despite the large variety in algorithms designed to perform clustering on data streams, it is far from being a solved field. Cluster analysis remains a foundational part of gaining useful knowledge from data, and so achieving high quality clustering

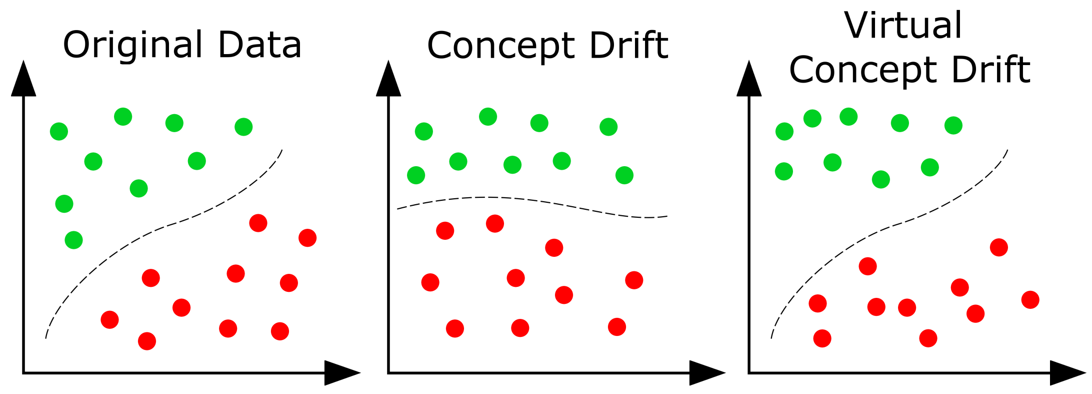


Figure 1.1: Real and virtual concept drift. Colours represent members of different classes.

is an important goal.

## 1.2 Problems With Stream Clustering

Originally, cluster analysis was a task to be performed on batches of data - algorithms mapping each data point in a static data set to a cluster, with the data being accessible multiple times and at will, and having a finite number of data points to process. As technology advanced and requirements changed the need arose for clustering to be performed on data streams.

Streams introduce multiple significant challenges compared to batch data, specifically:

- The data stream is potentially endless, so storing every data point is infeasible.
- Data points may be read from the stream only once, in the order that they arrive.
- The distribution of data points from the stream can change over time, requiring entirely different cluster mapping at different periods in the stream.

According to Schlimmer & Granger (1986), concept attainment is a field of machine learning in which one seeks to create a model learning to separate data points into specific concepts. These concepts represent a priori divisions of the data space into distinct categories. The idea behind learning these concepts and building a model to represent them is to successfully predict the concept to which a new unseen instance of data belongs to. Concept attainment includes supervised tasks like classification using training data, as well as unsupervised learning like clustering, in which no training data is used.

According to Gama et al. (2014), in machine learning, concept drift occurs when the underlying concepts change over time.

**Definition 1.2.1** *Concept drift occurs when the probability of a set of input variables $X$ being mapped to a target variable $y$ changes over time. In the context of data clustering, it is the probability that the observed data points, $X$ will be correctly mapped to the ideal ground-truth classes $y$ over time, which can be defined as:*

$$\exists X : p_{t_0}(X,y) \neq p_{t_1}(X,y), \tag{1.2}$$

where $p_{t_0}(X,y)$ represents the joint probability distribution of the data at time $t_0$ between the set of input data points $X$ and the distribution classes $y$, and likewise $p_{t_1}(X,y)$ represents the joint probability of points $X$ and classes $y$ at the later time $t_1$.

In reality we can only observe the unlabelled data points as they arrive from the stream - i.e. we only observe $p(X)$ over time (Gao et al., 2007). This makes it hard to determine whether the change is merely a change in sample population or real concept drift.

Real concept drift occurs when the probability of a given observation belonging to a given class changes over time - when $p(y|X)$ changes. Virtual concept drift, however, occurs when the population shifts, but the probability of an observation belonging to a given class doesn't change - when $p(X)$ changes, but $p(y|X)$ remains the same (Gama et al., 2014).

In the context of clustering, and this thesis, we are interested in real concept drift that affects the predictive power of clustering algorithms - changes in the underlying concepts in the stream over time. Since clustering is an unsupervised process we have

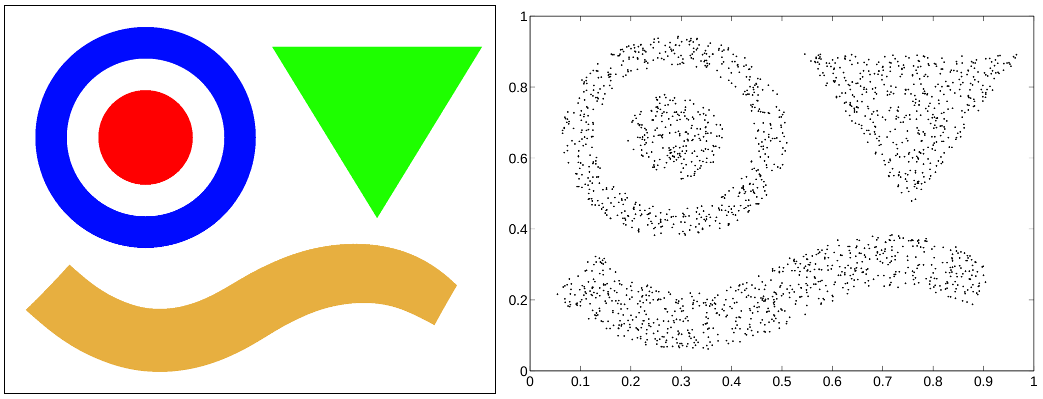


Figure 1.2: The underlying classes in a stream versus data points sampled from the same set of classes.

only the data points sampled from these underlying concepts available for observation. Adjusting to handle such changes in concept involve predicting changes in concept by examining the changes in population distribution of time.

Changes in data distribution, known as *stream evolution* or *concept drift*, is a particularly challenging property of data streams (Hulten et al., 2001). The data points from the data stream are sampled from an underlying set of classes, which ideally would map to clusters in the stream clustering context, as illustrated in Figure 1.2. The goal of cluster analysis is to discover and approximate these classes, and to map the given data points to clusters representative of them. Over time, these classes can change in density, move within the data space, merge, split, disappear, emerge, change size, rotate, or change shape. This would result in changes to the distributions of sampled data points. Such changes in underlying distribution can happen abruptly, or gradually over time, and both kinds of evolution can impact the integrity of clustering results if the algorithm is not designed to deal with it. In addition, noise in the stream can impact clustering in the same way that evolution does.

Stream evolution poses several challenges which have significant implications for stream clustering algorithms. Firstly, stream evolution means that older data may be less and less representative of the current data distribution as time goes on (Aggarwal, 2018). This can lead to algorithms defining clusters based on older data which is no longer relevant for the current state of the stream. This problem is often addressed using fading functions and decay of older data, in which newer data is prioritised in clustering and older data is weighted less, or gradually removed over time (Khalilian & Mustapha, 2010). A trade-off is inevitable between the consideration of historical data points in clustering decisions, and focusing on newer data points to reduce the possibility of basing clustering decisions off of data which is no longer relevant. This trade-off is often left to the user in the form of a parameter which affects the rate at which older data is made redundant.

Secondly, as the stream evolves over time the distribution of data can change in unpredictable ways such that the underlying classes are completely different at different times in the stream. Clustering algorithms typically rely on user-specific parameters which represent thresholds, or similar values, which affect how the algorithm groups data points into clusters. Setting these parameter values inappropriately can lead to clustering output which is meaningless, yielding little or no useful knowledge about the data. It is imperative that the parameters are set well in order to achieve useful clustering results. However, a major issue with stream clustering is that even if the initial parametrisation of the stream clustering algorithm results in high-quality clustering output at one point in the stream, concept drift may mean that those same parameters are no longer appropriate later in the stream.

## 1.3 Problem Statement

In light of the above issues, our purpose in this thesis is to achieve higher and more consistent quality clustering output in a streaming context. Particularly we wish to develop methods which respond to changes in the underlying data stream, and which are more robust to concept drift, requiring as little parametrisation by users as possible. Clustering is an unsupervised process, where often little, if anything, is known about the dataset prior to analysis. As such, algorithms which require users to specify parameters which are very sensitive to the data distribution are not robust with respect to unpredictable stream evolution. One of the greatest challenges in stream clustering is building algorithms without requiring ad-hoc critical parameters such as cluster number, density, grid granularity, graph connectivity, or window length (Silva et al., 2013). We propose, therefore, that it is vital for algorithms to consider concept drift with respect to algorithm parametrisation. It is vital to make these input parameters less sensitive, or, preferably, to reduce the number of input parameters a user must set, so as to achieve more robust and reliable clustering.

Because of the issues with stream clustering our motivation is to adapt to concept drift within the data streams in a stream clustering context. The first method for adaptation is by identifying when these changes occur. Such a process is known as *change point detection*. Firstly we wish to demonstrate that concept drift is detectable algorithmically by analysis of geometric and distributional properties of the stream clustering model. By employing change point detection methods we wish to identify when stream evolution occurs. Change point detection is itself an area of study in statistical analysis (Truong et al., 2018), and we wish to use it to identify when changes occur in arbitrarily dimensional data streams, as defined earlier. By analysing the geometric and structural properties of the multi-dimensional stream data we propose that these change points can be identified. Our goal is to create a method for reporting when change points happen in the underlying data as the stream progresses.

Secondly, we wish to improve the robustness of clustering algorithms. Specifically in this thesis we concentrate on the nearest-neighbour graph-based clustering approach. Typically this kind of clustering involves setting a $K$ connectivity parameter, which is sensitive and must be set correctly for useful clustering to occur. $K$-nearest neighbour ($K$NN) graph-based stream clustering algorithms are able to detect arbitrarily shaped clusters with arbitrary levels of density, this results in high quality clustering output at the expense of a higher level of computational complexity provided that the $K$ parameter is set appropriately. Our goal is to reduce the sensitivity of the critical clustering parameters used in this method of clustering, making it less sensitive to stream evolution. We do this by computing features based on the relative distances of edges connecting to nearby neighbours in the $K$NN graph structure, and using these features to inform clustering.

## 1.4 Contributions

The contributions of this thesis are:

- A thorough review of related background literature in Chapter 2, particularly in regards to current stream clustering approaches. We show their methods, categorise them by their approaches, and describe their various input parameters. We show that clustering relies on sensitive input parameters which must be set by users.
- A change point analysis algorithm for use in a multi-dimensional data stream in Chapter 3, which uses features computed from the nearest-neighbour structure of RepStream.
- A method for the analysis of the geometrical and statistical features in RepStream's graph structure to evaluate the effectiveness of the current $K$ connectivity parameter, and to indicate when and in which direction it should be changed over time in Chapter 4. This is achieved through the use of the *edge distribution score*, which is a statistical measure computed from the edges in the $K$-nearest neighbour structure in RepStream. The edge distribution score approximates how much the edge length distributions differ from the expected distribution, and allows us to infer properties of the local neighbourhood.
- An extension of RepStream in Chapter 4 which automatically varies its $K$ connectivity parameter in response to changes in the edge distribution score. We show how the thresholding parameter is relatively insensitive and robust compared to the sensitive $K$ parameter which must be set and remains static during the whole stream in the original RepStream version.
- A method for controlling the local connectivity of vertices in a nearest-neighbour sparse graph, in Chapter 5, through the analysis of the *skewness excess score*,

which quantitatively measures the relative skewness excess of the local neighbourhood in a nearest neighbour graph. We show how this measure can be used to approximate an appropriate level of connectivity of a vertex in its local neighbourhood.

- The RobustRepStream algorithm in Chapter 5, an extension of the RepStream algorithm which removes the need for the user to set the $K$ parameter entirely by using the skewness excess score. We show that this algorithm is robust across varied types of datasets, both on real-world and synthetic datasets.

## 1.5 Thesis Structure

Our thesis structure is as follows:

Firstly in Chapter 2 we present a thorough background and review of literature regarding stream clustering methods with respect to evolution of data streams. Specifically, we show the prevalence and importance of initial input parameters to the algorithms. Whilst the algorithms are often very different in which parameters must be specified by users, all the algorithms require some level of parametrisation and assumptions that must be made about the data. The papers reviewed in this chapter cover algorithms which deal with stream evolution via various techniques, and also covers a variety of different classes of clustering algorithms.

Having showed the wide variety of clustering approaches and their reliance on input parameters, in Chapter 3 we propose a method for detecting concept drift within data streams that evolve over time. Given that the evolution of concepts in the stream directly relates to the effectiveness of the input parameters, being able to measure and detect when this change occurs is paramount. We show that high-level computed features can be effectively used to identify when stream evolution takes place, and we experimentally show that our method performs better than other related stream change detection methods.

In Chapter 4 we build on the previous chapter by seeking to use similar computed features to produce more robust clustering. Given that stream evolution can be detected using these features, we hypothesise that they could also allow critical input parameters to be adjusted over time to produce better clustering. Thus, we present our extended version of RepStream, which uses a novel concept called the *edge distribution score* to determine an appropriate level of $K$ connectivity in RepStream's $K$-nearest neighbour sparse graph structure. Using the edge distribution score to automatically control the $K$ connectivity over time, we show that we can achieve clustering performance which is comparable to that of RepStream when the theoretical best $K$ parameter is set. We also show that our method achieves performance which is superior to other contemporary, sophisticated stream clustering approaches.

While allowing parameters to be automatically adjusted to more appropriate values over time in response to stream evolution is a good step, perhaps a better solution is reducing the number and sensitivity of parameters that clustering algorithms rely on. In Chapter 5 we introduce the RobustRepStream algorithm which performs stream clustering without the need for the user to set the sensitive $K$ parameter, which is the primary parameter of the RepStream algorithm. We introduce the concept of skewness excess, which is used by the proposed RobustRepStream algorithm to automatically determine appropriate levels of connectivity in its sparse graph structure without the need for a $K$ parameter. With the user no longer responsible for determining the critical $K$ parameter, we show that RobustRepStream produces consistently high quality clustering across all of the datasets we evaluate on, with no changes in input parameter values.

Chapter 6 concludes this thesis. We discuss the proposed methods in regards to handling data stream evolution and parameter sensitivity in clustering algorithms, as well as possible directions for future work.

# Chapter 2

# Background and Literature Review

Our research focuses on the adaptation of clustering algorithms to changes in a data stream over time, otherwise known as concept drift, or stream evolution. This chapter reviews the literature related to change detection in data streams, as well as approaches that have been used to adapt to concept drift in order to mitigate the problems it presents.

The literature reviewed in Section 2.1 examines clustering methods which analyse incoming data for distributional changes, as well as stream change detection methods. In Section 2.2 we examine various stream clustering algorithms, particularly in analysing the hyper-parameters used by each approach. This covers the different stream clustering approaches, including micro-cluster based, grid-based, and graph-based methods, as well as other approaches.

## 2.1 Change Detection Background

Data streams, as we discussed in the previous chapter, are characterised by evolving data. Data evolution poses major challenges to data mining algorithms, and so researchers have applied many techniques to address it (Khamassi et al., 2018). Concept drift is closely related to the idea of change detection in statistics, and so many techniques have been proposed which are inspired by change detection methods. Here we present an overview of such techniques, which seek to identify when change occurs, so that the knowledge can be used to better understand emerging concepts in the stream.

### 2.1.1 Approaches Including Change Detection

There are some clustering approaches which do use change detection mechanisms in their operation, either for the purposes of remaining consistent with newer data, or for discovering new concepts as they emerge. The grid-based algorithm clustering algorithm ExCC (Bhatnagar et al., 2014) uses a novel approach to handling drift in stream distribution. When new points are outside expected boundaries in ExCC they are added to a 'hold queue', which is periodically examined to determine whether change has occurred. If a sufficient number of new points are outside expected ranges then a 'change' is reported, otherwise it is counted as an outlier or short-term temporary drift. This method can be thought of as a form of detecting stream change, in the case of ExCC it is change within the position of new points on a grid. This is an example which does not use standard statistical methods of change detection. ExCC is described in more detail in Section 2.2.

An approach by (Masud et al., 2011) 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. This method detects the appearance of new classes in a stream, which is relevant to the field of cluster analysis and especially given that one of our primary goals is to adapt to emerging classes. Other topics related to change detection in data streams include recording and tracking change over time for the identification of temporal change (Dunham & Hahsler, 2011), or tracking change in a noisy stream through cluster density analysis (Nasraoui & Rojas, 2006).

### 2.1.2 Anomaly detection

Anomaly detection is a related field which seeks unexpected or anomalous inputs which may be used for data streams. (Chandola et al., 2009) categorise anomaly detection approaches into the following categories. Classification-based approaches construct a classifier using training data, then classifies new data points into one of the learned classes, examples of which are provided by (Janssens et al., 2009). Nearest-neighbour based approaches use a distance measure between data points and either take the distance to its $k^{th}$ nearest neighbour, or which calculate a density based on distance to neighbouring points.

An example is iNNE (Bandaragoda et al., 2014) which attempts to isolate new points from its nearest neighbours to calculate an anomaly score. Clustering-based approaches use clustering algorithms to group normal data into a series of clusters, and new data points are considered to be anomalous or not depending on whether they match an existing cluster for instance (Rajasegarar et al., 2014) uses clustering algorithms on distributed sensor networks and examines the inter-cluster distance for anomalous data.

Statistical anomaly-based techniques compare new points to a model and generate either an anomaly score or perform a hypothesis test to probabilistically determine when change occurs (Soule et al., 2005). Some more novel techniques use other concepts such as spectral methods, 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. These algorithms are somewhat similar in concept to change detection, concentrating on unexpected transient changes which differ from what is considered to be 'normal' behaviour. Anomaly detection can, in a sense, be considered as a more specific and constrained version of the change detection methods we are exploring in this work.

### 2.1.3 Stream Change Detection Methods

There are relatively few examples in existing literature of algorithms which deal with change specifically in a streaming context. However, there are some which operate on streams, either of transaction data or a series of time-ordered multidimensional vectors.

STREAMKRIMP (Van Leeuwen & Siebes, 2008) is an algorithm which is used to detect changes in data streams involving transaction and item data. STREAMKRIMP constructs a Minimum Description Length code table as it reads the stream. This is used to compress the data as it arrives in the stream. When the distribution of transactions in the stream changes the compression rate of the code table also changes. This prompts STREAMKRIMP to make a new MDL code table, which indicates a change in the stream. This approach is similar to our goal of detecting distributional stream change, however it is designed to work on transaction data rather than multidimensional vector streams.

Another method by (Qahtan et al., 2015) uses a principle component analysis (PCA) based approach to detect changes in data streams. Specifically, it seeks to detect changes in high dimensional data by projecting down into multiple one-dimensional data streams, using the principle components. These projected streams are then analysed for change from a given reference window to a test window comprised of new points. This algorithm performs well when on detecting change on high dimensional data sets.

## 2.2 Stream Clustering Algorithms

There has been a wealth of research regarding stream clustering approaches, particularly in the area of increasing efficiency and accuracy of clustering output. A common theme, however, is the reliance on input parameters - set values which affect the clustering output. Since clustering is an unsupervised exploratory data mining process, these parameters can't be tuned in response to training data in the same way that supervised classification algorithms can. Instead, the users must rely on their own intuition about the data structure, or guidance from the algorithm's author.

Stream clustering algorithms are divided into a range of different general approaches which we present here.

### 2.2.1 Micro-Cluster/Density-Based Clustering

CluStream (Aggarwal et al., 2003) is an early example of a distance-based micro-clustering framework for use in stream clustering. CluStream uses Cluster Feature Vectors (CFVs) to maintain information about groups of data points, which summarises the data of multiple data points into a single structure. The CFVs are made up of a tuple, which includes the sum of the data values for each associated data point in each dimension, the sum squared of the data values for each associated data point in each dimension, as well as the sum and sum squared of the time-stamps of each data point associated with the CFV. This type of micro-cluster has desirable properties, being both additive, so that CFVs can be easily merged, and also being subtractive such that a snapshot of a CFV at a previous time can be subtracted from the current time to yield data about how the CFV has evolved. CluStream maintains snapshots of its CFVs in a pyramidal way to make use of this property - with more recent snapshots being kept than older ones. A parameter $\alpha$ affects the frequency of snapshots, and another parameter $l$ determines how many snapshots are stored. New points incoming from a stream are added to existing CFVs if they are within a boundary defined by a parameter $t$, or become new CFV micro-clusters otherwise. A relevance threshold $\delta$ determines when an existing CFV can be removed and replaced by a new one. The CFV micro-clusters are used in an offline stage to build clusters with a variant of the $k$-means clustering algorithm that treats the CFV micro-clusters as pseudo-points. The micro-cluster structure is a very common concept which is used in many other algorithms.

The DenStream algorithm (Cao et al., 2006) uses a variant of the micro-cluster concept to do density-based clustering on stream data. DenStream handles streaming data using a damped-window model, meaning that the weight of each individual points decays exponentially over time according to a decay parameter $\lambda$, which makes the algorithm biased towards more recent data. DenStream defines three types of micro-cluster - the Core Micro-Cluster (CMC), the Potential Core Micro-Cluster (potential-CMC), and the Outlier Micro-Cluster. A newly added data point from a stream becomes part of an existing micro-cluster if it is within a distance defined by a parameter $\varepsilon$, or becomes a new outlier micro-cluster if it can't be added to an existing one. A micro-cluster becomes a CMC if its weight is greater than or equal to an input parameter $\mu$, or otherwise is a potential-CMC if its weight is greater than $\beta\mu$ where $\beta$ is an input parameter $0 \leq \beta \leq 1$. Micro-clusters which have weight less than $\beta\mu$ are considered to be outlier micro-clusters. Micro-clusters are clustered together using a variant of DBScan, in which micro-clusters are regarded as virtual points. CMCs which are less distance from each other than the sum of their micro-cluster radii are grouped together into the same cluster, while Potential-CMCs follow this same rule but can only be grouped with CMCs. Outlier micro-clusters are not included in any cluster, and are maintained in a separate memory space. This density-based clustering approach allows the algorithm to locate arbitrarily shaped clusters, unlike CluStream and SWClustering which can find only hyper-spherical clusters.

High dimensionality can make traditional clustering algorithms produce less useful results, since a common distance metric used is Euclidean distance, and the HPStream algorithm (Aggarwal et al., 2004) seeks to address this. Data sets with high levels of dimensionality suffer the so-called curse of dimensionality, in which distance functions become less and less useful in higher dimensions. HPStream projects clusters into fewer, more relevant dimensions in order to perform clustering. HPStream initialises $k$ clusters using the $k$-means algorithm, and cluster dimensionality is calculated based on these cluster. Dimensionality for these clusters is defined with a bit-vector, in which the input data is normalised and the $l$ dimensions with the lowest radii for that cluster are chosen to be projected into. The parameter $l$ corresponds to an expected dimensionality for clusters in the data-space. Distance calculations for each cluster are done using that cluster's dimensionality bit vector - i.e. the clusters only consider the dimensions which are marked with a 1 in the dimensionality bit vector. This reduces the effects of high dimensionality on the distance calculations. The initial cluster centres are moved and dimensionality bit vectors are recalculated until they converge. Subsequent points which are added to HPStream are added to the nearest cluster, where the distance is projected using the bit vector of each cluster, if it is within a distance defined by a spread factor $r$, otherwise a new cluster is created. HPStream differs from the previously mentioned algorithms in that it works in an incremental online manner, having no separate offline clustering phase. Clusters monotonically decay, like cluster structures in other methods, using a decay function defined by a decay factor $\lambda$.

The BEStream algorithm (Wattanakitrungroj et al., 2018) takes the micro-cluster concept and extends it by introducing the idea of elliptic micro-clusters. In BEStream clusters are in the form of hyper-dimensional ellipses arranged along the eigenvectors of the data distribution, which differs from the standard hyper-spherical micro-clusters used in other algorithms. Additionally, BEStream can handle individual data points from a data stream, or batches of data at a time. Data points are added into existing micro-clusters, with their elliptical shapes being adjusted if necessary to fit the data. Data is captured into micro-clusters, which have a radius according to a parameter $\xi$ affecting the elliptical micro-cluster's size. To merge micro-clusters together a direction threshold $\theta$ and a distance threshold $\Delta$ are used to determine when micro-clusters can safely be combined into a single elliptical micro-cluster. In the macro-clustering phase a density threshold $\tau$ is used, and overlapping clusters with a density at least $1-\tau$ similar may be considered part of the same cluster

Whilst micro-cluster and grid-based approaches are common in stream clustering there are problems that are associated with them. The DBStream algorithm (Hahsler & Bolaos, 2016) is designed to counter the common assumption in grid and micro-cluster-based algorithms that points have an even distribution within grid cells and micro-clusters. The assumption by other algorithms can not capture separation that occurs within micro-clusters or grid cells. Grid and micro-cluster-based algorithms are usually unable to capture separation that occurs within micro-clusters or grid cells because of an assumption of an even distribution within grid cells and micro-clusters. DBStream (Hahsler & Bolaos, 2016) addresses this problem by allowing data points to be shared between micro-clusters, having the micro-clusters overlap in order to retain more data about the distribution of the points represented by the micro-clusters. Newly added points become part of a micro-cluster if they fall inside the micro-cluster's radius, defined by a radius parameter $r$, otherwise they become the centre of a new micro-cluster. New data points cause micro-clusters to shift position towards the newer points. Data about the number of data points that are shared in the intersection between micro-clusters is recorded for clustering purposes. Reclustering in the offline phase is done using the shared density information. If the shared density exceeds a given threshold $\alpha$ then the micro-clusters merge into the same cluster. Micro-clusters fade using a fading function and parameter $\lambda$ and micro-clusters are removed periodically every $t\_gap$ time steps, if the micro-cluster is below $w\_min$ weight. This shared density model helps to avoid the problem of uneven distributions inside micro-clusters, as even nearby micro-clusters will not merge if they don't have a large shared density. Overlapping micro-clusters with few, or no shared data points imply a level of separation between the micro-clusters which is not captured by traditional micro-cluster models.

Another algorithm which builds on CluStream and uses a similar micro-cluster stricture is SWClustering (Zhou et al., 2008). The SWClustering algorithm uses Temporal Cluster Features (TCFs), that contain similar information to the tuple used by CluStream - sum and sum squared of data values for each associated data point in each dimension, number of records, and most recent time-stamp. TCFs are stored in an exponential histogram, such that more recent records are stored with more detail and granularity than older TCFs. The older TCFs become merged together if the current exponential bucket exceeds $\frac{1}{\varepsilon}+1$ records, where $\varepsilon$ is a limiting parameter. Merges are cascaded through the exponential levels, and the last TCF is deleted if its time-stamp is no longer one of the most recent. These Exponential Histogram of Cluster Features (EHCF) act like micro-clusters, and new data points are added to their nearest EHCF based on a radius threshold $\beta$, and at most $N$ EHCFs can be in memory. EHCFs are deleted or merged when the number exceeds the threshold. The EHCF structure allows more granularity on more active clusters over time than the pyramidal snapshots used in CluStream. SWClustering, like CluStream, uses the $k$-means algorithm to cluster the EHCFs as pseudo-points, where each EHCF is weighted based on the number of records it contains.

### 2.2.2 Grid-Based Clustering

While micro-clusters are a popular approach in clustering, an alternative method is the grid-based approach, like that used in D-Stream (Chen & Tu, 2007). The D-Stream algorithm partitions the data space of the input stream into a fixed granularity grid, in which the input data is normalised to $[0,1]$ and each dimension is partitioned into even segments with a length determined by a parameter $len$. A value of $len = 0.02$ corresponds to each dimensions being partitioned into 50 even segments, with in total $50^d$ cells, where $d$ is the number of dimensions in the data. Like many other stream clustering algorithms D-Stream uses a parameter $\lambda$ to control the decay of data, and bias the algorithm towards newer data. Each cell maintains a characteristic vector which contains data on its last time of update, last grid density, the class label, and whether the cell is a sporadic or normal cell. Each time a data point is added to the grid it increases the weight of a cell. When a cell reaches a weight above a user defined parameter $C_m$ it becomes a dense cell, if the weight is lower than $C_m$ but higher than another parameter $C_l$ it is a transitional cell, otherwise the cell is called a sparse cell. Periodically sparse cells may become marked as sporadic if their weight is below a threshold set by the parameter $\beta$, if a sparse cell has already been marked as sporadic and remains below the required weight threshold on the next periodic check then it is removed from memory to save space. In the offline clustering phase, adjacent dense cells become merged together and are assigned the same cluster label, working similar to how core micro-clusters work in DenStream. Similarly, transitional cells are grouped into the same cluster as adjacent dense cells.

Density and grid-based techniques are used in the (Dense Units Clustering) DUCStream algorithm (Gao et al., 2005) to form an online incremental clustering method, which is in contrast to many methods which separate maintenance and clustering phases into online and offline components respectively. DUCStream partitions data into non-overlapping hyper-rectangles - a multi-dimensional grid, in which individual cells are referred to as local units. The density of a unit is equal to the number of data points contained within that unit, and the unit is considered to be a dense unit if its relative density exceeds a given density threshold $\gamma$. A unit is considered to be a local dense unit if $den(u)/m(t-i+1) > \gamma$ where $den(u)$ is the density of the given unit, $m$ is the size of a chunk of data points, $t$ is the current time, and $i$ is the time when the unit $u$ started being maintained. In this manner, even if many data points belong to a given unit it still may not qualify as a local dense unit if relatively more data points belong to other units in the same chunk. Local dense units are used for clustering, with dense units being combined with other adjacent dense units to form clusters.

The PatchWork algorithm (Gouineau et al., 2016) is another grid-based clustering algorithm designed to be easy to deploy in a distributed way. This allows for linear horizontal scalability. PatchWork works by inserting data points into a grid, in which the grid's granularity is controlled by a parameter $\varepsilon$. Grid cells that contain data points are sorted by density, and the cell with the highest density is selected for processing. Cells nearby that cell are clustered together if their density is greater than a fraction $ratio$ of the original cell's density. This process is repeated, with the next highest cell being selected for clustering, until all cells containing data points have been processed. Optional $minPoints$ and $minCells$ parameters are available to filter out clusters which are too small.

Similar to grid-based approaches the MR-Stream algorithm (Wan et al., 2009) keeps multiple granularities of data grids in memory to allow for multiple resolution analysis and clustering of a data stream. The data space is initially partitioned into $2^n$ cells, where $n$ is the dimensionality of the data stream. At each successive level of resolution the cells are further split into an additional $2^n$ cells. The total number of cells at each level is $2^{nh}$, where $h$ is the current level of the multi-resolution partition. A parameter $H$ defines the upper limit to the resolutions stored by the algorithm. Data points are added into the grid at all resolutions, adding to the weight of one cell at each resolution that contains the data point. MR-Stream uses a fading function for records added to a cell, with the fading factor $\lambda$ controlling the speed of the decay of the weight. Cell density at each resolution is the weight of a cell divided by the volume, where the volume of each highest-level cell is 1, and the total volume of every cell in the data space is $2^{nH}$. Like D-Stream, MR-Stream uses the parameter $C_H$ as a threshold for the minimum density a cell must be to be considered a dense cell. Similarly the $C_L$ algorithm is the upper-bound threshold for when a cell is considered a sparse cell. Cells between these thresholds are called transitional cells. At each resolution, clusters are formed by linking nearby cells together. A cell must be within a distance $d$ from a dense cell to be part of the same cluster. The value of $d$ is defined by a parameter $\varepsilon$ where $d = \varepsilon^{2-H}$. The algorithm can filter out smaller clusters at finer granularities by labelling them as noise clusters. A cluster is considered to be a noise cluster if its total weight is less than a parameter $\beta$ or if its volume is less than a parameter $\mu$.

Another example of grid-based clustering is the Exclusive and Complete Clustering (ExCC) algorithm (Bhatnagar et al., 2014). ExCC is another fixed granularity approach, but unlike D-Stream its only input parameters are grid granularity for each dimension. Cells in the grid are stored in a tree structure, with each leaf storing the number of data points in the cell, time of the first point to arrive, and the time of the most recent point. The average inter-arrival time of points for each cell can be calculated based on the number of points, and the two previously mentioned stored timestamps. ExCC prunes old cells on calls for clustering. Prior to clustering cells are pruned from the tree structure if they have not been updated in more than the average inter-arrival time. This helps reduce the memory usage of the algorithm, and makes sure older less relevant data is not considered in clustering. Cells which are not pruned at a clustering call are merged together into clusters if they are adjacent. Cells are considered to be dense cells when their weight surpasses an internal threshold $\psi$, which is defined as $\psi = \lceil \frac{\mu}{ln(g+d)} \rceil$ where $\mu$ is the average number of points in each cell, $g$ is the average granularity of the dimensions, and $d$ is the number of dimensions in the data. Cells which are above this computed threshold form the core of clusters in ExCC, with adjacent dense cells being grouped together. To satisfy the goal of complete clustering, cells which are not dense are added onto adjacent clusters, such that all data points belong to a cluster. ExCC also has a mechanism for handling data drift outside of defined maximums and minimums in each dimensions. When a data point from the stream is outside the bounds of a dimension it is considered to be an anomalous point and is added to a hold queue. This hold queue is periodically processed, and if the number of anomalous points exceeds the number of points in the boundary cells, then the grid's boundaries are expanded to deal with the data drift.

### 2.2.3 Graph-Based Clustering

Another approach is the graph-based clustering approach, in which data points, or micro-clusters, and connected into a tree or graph structure. There are many clustering algorithms that use graph-based approaches, however methods which require nearest neighbour searches can be computationally expensive. Minimum spanning trees, are a popular structure for clustering algorithms because they start from a minimal connectivity between data points, and therefore are efficient with respect to producing clusters. Using a minimum spanning tree (MST) turns clustering from a problem of grouping points together into a problem of splitting the tree into segments representing clusters. A successful graph-based approach requires consideration about how to be more efficient than a strict nearest neighbour search including all data points. The PASCAL algorithm (Cagnini & Barros, 2016) is a graph-based batch clustering algorithm which uses a minimum spanning tree in its clustering. PASCAL is introduced as a parameterless clustering method, working on a static distribution of data points, and seeks to find clusters by breaking the MST into pieces which correspond to clusters. The MST contains exactly $N-1$ edges between nodes, which is a significant reduction on the number of edges in a $k$-nearest neighbour graph, and reduces the clustering problem to one of finding which edges to remove from the MST to identify clusters. The MST is mapped onto a direct probabilistic graphical model, in which the probability of objects belonging to the same initial cluster is given by a function dividing euclidean distance by the weight of edges in the MST. The probabilities in the graphical model are mapped into a univariate distribution which PASCAL samples from to identify which pairs in the MST are **must-link** and which pairs are **cannot-link**. PASCAL then uses an evolutionary algorithm, iterating over a number of generations, removing the worst must-link pairs from the population according to the density-based clustering validation criterion as a fitness function. The pairs are removed from the population, and new pairs are added from the GM until the solution converges to a peak in the fitness value. This approach makes use of evolutionary convergence and also a MST graph-based data structure to find arbitrarily shaped clusters in a static non-streaming dataset. The approach is parameterless in the clustering, except in the evolutionary estimation of density algorithm used to converge towards the solution, which requires several hyper-parameters.

The HASTREAM algorithm (Hassani et al., 2014) uses a hierarchical model for the MST to provide multi-resolution clustering. In the online phase HASTREAM maintains a series of micro-clusters, in which the radius is defined as $\sqrt{\frac{|\overline{CF_2}|}{w} - (\frac{|\overline{CF_1}|}{w})^2}$ and the centre is $\overline{CF_1}$, where $w$ is the weight of the points $\overline{CF_1}$ is the linear weighted sum of the data points and $\overline{CF_2}$ is the weighted sum of the squared data points. This is similar to the definition of micro-clusters in CluStream, but with the radius defined by the distribution of the data points rather than a parameter. Data points are added to existing micro-clusters if possible and the weight is updated based on the number of added points. The clusters decay if no new data points are added however, with a fading function controlled with a parameter $\lambda$. The offline phase of HASTREAM requires an input parameter *minPts*, which acts as a density threshold. During the offline phase, micro-clusters are considered as vertices in a graph, with a core distance defined as the Euclidean distance to that vertex's *minPts*'th nearest neighbour. The mutual reachability distance between two micro-clusters is defined as the maximum of the Euclidean distance, and each micro-cluster's respective core distance. A mutual reachability graph is constructed, which is a complete graph, with the weight of each edge being the mutual reachability distance of each pair of vertices. A minimum spanning tree is constructed using this mutual reachability graph. A hierarchical dendrogram is then constructed from the MST, the top level containing the entire MST and all micro-clusters in one cluster, then lower levels splitting the MST by removing the longest edge, which splits the dataset into its component sub-clusters. At any level of the dendrogram, if the total weight of the subcomponent does not exceed t *minPts* parameter then it is considered to be noise. Using this dendrogram, clusters at various hierarchical levels can be extracted manually, or HASTREAM can produce a flat clustering by itself by using a concept called cluster stability. Cluster stability is a function derived from the difference of a maximum and minimum density thresholds for which the sub cluster will continue to exist, multiplied by the weight of the cluster.

The clustering used as the final flat clustering is the set of non-overlapping clusters from the dendrogram which has the highest sum of cluster stability. Intuitively, the sub clusters which are most stable through a range of threshold values are likely to be well separated from the rest of the data, so these are logical choices for the flat clustering.

Nearest neighbour graph-based clustering algorithms are computationally intensive due to having to update and maintain edges connecting to nearest neighbours for each vertex in a graph. SNCStream (Barddal et al., 2015) is an algorithm which uses social network principles to make this connectivity maintenance more efficient. SNCStream uses a $K$-nearest neighbour clustering approach, in which each vertex has edges connecting to the $K$ neighbours which are closest using a given distance measure. Instead of using the strict nearest neighbours, the maintenance phase uses two-hop neighbours - that is, it searches only data points that connect to its current neighbours when searching to update its list of nearest neighbours. This process is not guaranteed to result in an exact $K$-nearest neighbour graph, but it is likely to be similar to the true KNN graph, and requires that at *most* $K^2$ neighbours be searched instead of all vertices in the graph. The vertices in SNCStream are treated like micro-clusters similar to DenStream after an initial window size $N$ points have been processed. New data points are added to existing micro-clusters, or become new outlier micro-clusters if no existing micro-cluster is nearby. Similar to DenStream, a micro-cluster must have sufficient weight to become a potential micro-cluster, and be added to the graph. The parameter $\beta$ is the same as in DenStream and $\psi$ replaces the $\mu$ parameter for determining when a micro-cluster is an outlier. The $\varepsilon$ parameter determines the radius of micro-clusters, and the $\lambda$ parameter controls a fading function. The benefit of SNCStream is the increased efficiency over other graph-based clustering methods, resulting in faster neighbour searches at a small risk of incorrect graph structure.

The FastAP algorithm (Sun et al., 2017) is another algorithm which uses graph-based approaches in its clustering method. FastAP was designed with the idea of making affinity propagation faster and more scalable by reducing the size of the similarity matrix used during the message-passing phase. To do this, FastAP uses a two-stage algorithm which consists of a *compression* and *sparseness* phase. The compression stage reduces the similarity matrix that affinity propagations use from a $N^2$ matrix to being a $Ng$ matrix, where $N$ is the number of data points in total, and $g$ is the number of exemplars selected during the compression phase. The compression is done by treating each point as a potential exemplar, and then iteratively merging each point with its nearest neighbour, creating a micro-cluster, until the given compression ratio is reached. The second phase is the sparseness phase which uses a $K$-nearest neighbour grap-based approach at further reducing the similarity matrix. This occurs by reducing the matrix based on the $K$-nearest neighbour connectivity of the potential exemplars chosen in the previous stage. This reduces the size of the similarity matrix to $Nk$, where $k$ is the level of connectivity in the $K$-nearest neighbour sparse graph. Affinity propagation message passing is done on this greatly compressed similarity matrix, reducing the computational complexity whilst producing clustering output of comparable quality to standard affinity propagation approaches.

### 2.2.4 Other Novel Clustering Methods

Affinity propagation is used in multiple clustering algorithms, and the ADStream algorithm (Ding et al., 2016) combines this with a micro-cluster approach which extends DenStream. ADStream is split into an online and offline component, performing micro-cluster creation and maintenance in the online component, and data clustering in the offline component. The online component uses affinity propagation to turn data points into micro-clusters for later reclustering. When the algorithm starts an initial number of data points are used to build a similarity matrix. An affinity propagation algorithm is then used to determine an initial clustering for online maintenance. These clusters become micro-clusters, where the weight is determined by the number of points the micro-cluster contains, with point weights decreasing according to a fading function using a parameter $\lambda$. When the similarity of points - the sum of the responsibility and availability in the affinity propagation component - exceeds a threshold parameter $\xi$ then it becomes a candidate density micro-cluster, or a full density micro-cluster when the weight exceeds a threshold $\varepsilon$. These density micro-clusters are used in an offline phase, as in DenStream, to build clusters.

While many algorithms require parameters to complete their clustering, as shown in Table 2.1, there are some approaches that are genuinely parameterless. One such algorithm is the DeBaRa (Schneider & Vlachos, 2013) algorithm, which combines a density-based approach in combination with projecting data onto random lines. This approach is not used for data streams, however it allows data to be clustered without needing parameters to be set by the user. The DeBaRa algorithm works initially by partitioning the data point into smaller subgroups. It does this by creating a random one-dimensional line through the dataset and projecting the data onto that line, by taking the shortest distance from the point to the line. Once the points have been projected onto the line, a random point is selected and all points on one side of the line become one subgroup, and the rest of the points become another subgroup. The partitions are partitioned like this repeatedly, being projected onto new random lines, until the subgroups are all a minimum set size. A point's neighbours are considered to be the points closest to them on the projected lines, and a number $dPts$ of them are used for clustering calculations. A fraction $f$ of those neighbours are used to calculate an average distance for clustering purposes. Points within these partitioned are labelled as merge candidates when two points are mutually within the minimum of the two points' merging distances on the given random projection. This restricts which points may be merged together in the clustering phase. Schneider and Vlachos propose this approach to improve the efficiency of the OPTICS algorithm, but also propose a parameterless clustering algorithm to complement their random-projection preprocessing steps. This method involves gradually increasing the $dPts$ parameter. For a static value of $dPts$ it identifies low-density points - points which are lower density than all surrounding neighbours. All other points points which are labelled as non-separating merge with all nearby points, while the low-density points greedily merge with the neighbour of maximum density. This method requires no input value for $dPts$ and splits all points into clusters.

Inspiration for novel solutions to problems can come from many places, and FlockStream (Forestiero et al., 2013) is one such algorithm which uses a biologically inspired model to perform clustering. The FlockStream algorithm extends the Multiple Species Flocking Model into the data clustering realm, using emergent behaviours of flocking agents to locate arbitrarily shaped clusters. The algorithm represents data points as boids, or agents, which follow a set of rules based on the Multiples Species Flocking Model. Specifically these rules are:

- Boids will aim to unify their alignment - direction of movement - with members of the same species.
- Boids will aim to more towards members of the same species.
- Boids will aim for a minimum level of separation from other members so that flocks don't become too dense.

These boids are placed randomly onto a hypothetical 2D grid. Each boid has a visibility radius around it, and it will follow the flocking rules for any member of the same species within this radius. Two boids in FlockStream are considered to be the same species/similar if the data points they represent are within some distance threshold $\varepsilon$ of each other in the data-space. Using these rules in the hypothetical 2D grid the boids align themselves and form flocks with similar boids. Flocks can become micro-clusters of different types. If the weight of a flock is higher than $\mu$ it corresponds to a core-micro-cluster. Weights decay over time according to a decay function and a decay parameter $\lambda$. Micro-clusters with weights lower than $\mu$ but higher than $\beta\mu$ where $0 < \beta < 1$ are potential-micro-cluster, and micro-clusters with lower weight are called outlier-micro-clusters. This groups boids as single entities to reduce storage and processing requirements. Micro-clusters act in the same way boids do, referred to as swarms, and flocks of swarm agents correspond to clusters. One issue with this algorithm is the random nature of its clustering. The algorithm runs through many iterations to converge into a stable solution, however it is not guaranteed to converge.

Other clustering methods which have unique approaches to handling one or more aspects of stream clustering, for example SyncTree Shao et al. (2019) which builds on the concept of micro-clusters by treating them as phase oscillators. Nearby objects have their phases synchronised with each other, which gives a method of clustering based on the synchronicity of objects in a local area.

### 2.2.5 Parameters In Clustering

We note importantly that the data stream clustering algorithms surveyed above take as input a set of parameters which are set at initial runtime and which are fixed through the length of the algorithm's execution . For example, ExCC (Bhatnagar et al., 2014) uses a fixed grid granularity parameter as its primary input, much like the PatchWork algorithm (Gouineau et al., 2016) which uses $\varepsilon$ as its grid granularity parameter as well as a density threshold tuning parameter. DUCStream (Gao et al., 2005) is a density-based method which requires a density threshold $\gamma$ as its primary parameter. FlockStream (Forestiero et al., 2013), which clusters based on biologically inspired methods, requires a distance threshold $\varepsilon$, as well as $\mu$ micro-cluster and $\beta$ distance thresholds. There are also some algorithms such as (Ackermann et al., 2012) which use an implementation of $k$-means that require the number of clusters, $K$, as an input parameter. Additionally, the STRAP (Streaming AP) algorithm (Zhang et al., 2008) applies the affinity propagation technique in a streaming context. STRAP uses message passing to find a number of $K$-centres, using an energy minimisation approach. While this method does use a number of parameters, it is not necessary to specify the number of $K$-centres or clusters that are to be found by the algorithm.

RepStream (Lühr & Lazarescu, 2009), the base algorithm we have chosen for our work, is also an example of this, as its primary input parameter is the $K$ value, representing the number of outgoing edges from each vertex in its sparse-graph data representation. Some of the earliest stream clustering algorithms, including STREAM (O'callaghan et al., 2002) and CluStream (Aggarwal et al., 2003) also rely on critical input parameters which must be set by the user.

Table 2.1 shows the list of primary parameters used by the various clustering methods we have examined. Many of these parameters are similar between approaches, for example the use of a grid granularity parameter in grid-based approaches like D-Stream (Chen & Tu, 2007) and ExCC (Bhatnagar et al., 2014), or density thresholds used in density-based methods such as in HAStream (Hassani et al., 2014) and DB-Stream (Hahsler & Bolaos, 2016).

The main challenge is that some of the input parameters are very sensitive with respect to the quality of the clustering output produced by the algorithms. The reliance on parameters is considered to be an extremely important problem in stream clustering (Silva et al., 2013). It is entirely up to the user to specify input values that are suitable for a given problem. However, given that data streams are potentially very dynamic, setting optimal parameters is a non-trivial task. Due to concept drift, even if a set of values are suitable at the beginning of a data stream, it is not certain that they may be appropriate for algorithm later when the stream evolves significantly. It is therefore desirable to have a scheme that automatically sets suitable values for important parameters, or which otherwise minimises the sensitivity of critical user-set parameters. This is exactly what we aim to do in this work. While plenty of literature focuses on methods which improve the quality of algorithms overall, there is a lack of literature which concentrates on the sensitivity of algorithms to parameter values, and in increasing algorithm robustness with respect to those parameters. Our methods here differ from the prior work in that we attempt to address the problem of algorithm robustness and sensitivity to parameters.

Whilst there are no stream clustering algorithm that automatically updates its initial parameters, we are aware that there are few notable *non-stream* clustering algorithms that do not require parameters. For example, TURN* (Foss & Zaiane, 2002), a two-part clustering algorithm in which TURN-RES produces clustering output and statistics, while TurnCut attempts to adjust the resolution of TURN-RES using these statistics. Tseng and Kao also propose a method for domain-specific parameter selection of a clustering method used in gene expression analysis (Tseng & Kao, 2005). This method uses on-the-fly validation techniques, namely Hubert's $\gamma$ statistic to optimise the clustering. The DeBaRa algorithm (Schneider & Vlachos, 2013) is another parameter-free

Table 2.1: Table showing the input parameters used for each algorithm.

| Algorithm | Params | Algorithm | Params |
|---|---|---|---|
| CluStream (Aggarwal et al., 2003) | $t$ Distance Param<br>$\delta$ Decay Param<br>$\alpha, l$ Snapshot Params<br>K-Means reclustering | SWClustering (Zhou et al., 2008) | $\beta$ Density Param<br>$\varepsilon, N$ - Window Params<br>K-Means reclustering |
| DenStream (Cao et al., 2006) | $\mu$,$\beta$ Density Params<br>$\varepsilon$ Distance Param | HPStream (Aggarwal et al., 2004) | $\lambda$ Decay Param<br>$r$ Distance Param<br>$l$ Projected Dimensions |
| DBStream (Hahsler & Bolaos, 2016) | $\alpha, w\_min$ Density Params<br>$\lambda, t\_gap$ Decay Params<br>$r$ Distance Param | BEStream (Wattanakitrungroj et al., 2018) | $\Delta, \tau$ Density Params<br>$\lambda$ Decay Param<br>$\xi$ Distance Param<br>$\theta$ Direction Param |
| HASTREAM (Hassani et al., 2014) | $minPts$ Density Param<br>$\lambda$ Decay Param<br>Hierarchical Clustering | PASCAL (Cagnini & Barros, 2016) | Evolutionary EDA<br>hyper-Params |
| SNCStream (Barddal et al., 2015) | $\psi, \beta$ Density Params<br>$\lambda$ Decay Param<br>$\varepsilon$ Distance Param<br>$K$ - Graph Connectivity | FastAP (Sun et al., 2017) | $k$ - Graph Connectivity<br>$Cr, Sr$ Graph Compression Params |
| PatchWork (Gouineau et al., 2016) | $ratio, minPoints, minCells$<br>Density Params<br>$\varepsilon$ - Grid Granularity | MR-Stream (Wan et al., 2009) | $C_H, C_L, \beta, \mu$ Density Params<br>$\lambda$ Decay Param<br>$\varepsilon$ Distance Param<br>$H$ Hierarchical Limit |
| DUCStream (Gao et al., 2005) | Grid Granularity<br>Data Chunk Size<br>$\gamma$ Density Threshold | ExCC (Bhatnagar et al., 2014) | Grid Granularity in<br>all dimensions |
| D-Stream (Chen & Tu, 2007) | $C_m, C_l, \beta$ Density Params<br>$len$ Grid Granularity<br>$\lambda$ Decay Param | ADStream (Ding et al., 2016) | $\xi, \varepsilon$ Density Params<br>$\lambda$ Decay Param |
| DeBaRa (Schneider & Vlachos, 2013) | $dPts$ Density Param<br>$f$ Distance Param | RepStream (Lühr & Lazarescu, 2009) | $\alpha$ Distance Param<br>$k$ - Graph Connectivity<br>$\lambda$ Decay Param |

algorithm. It was proposed alongside a variation of the OPTICS density-based algorithm and clusters based on the relative density of points, allowing more dense points to merge clusters together, while low density points (points that have lower densities than their surrounding data points) may only join existing clusters. While this approach uses no data-dependent input parameters, the algorithm operates by incrementally adjusting an internal distance parameter to produce different granularity of clustering results. There are also previous works which have attempted to automatically select parameters, for example Maier et al. (2007) who attempt to prove bounds for symmetric and mutual $K$-NN graphs using probabilities based on probability of between-cluster connectivity. Mohd et al. (2012) propose a method for automatically determining the number and initial position of $k$-means centroids by measuring the maximum distance between data points. These methods provide insight into ways that automatically selecting parameters can be addressed, but apply to specific methods which are not applicable to an evolving stream clustering context.

To the best of our knowledge there is no prior work on stream clustering algorithm dynamic parameter selection and keeping the value updated over time as we present in Chapter 4, nor in using self-arranging sparse graphs to perform graph-based clustering as in Chapter 5.

## 2.3 The RepStream Algorithm

In this thesis we extend the RepStream algorithm (Lühr & Lazarescu, 2009), which is a combination density and graph-based stream clustering algorithm. The RepStream algorithm has been shown to perform very well against other similar algorithms, outperforming them in terms of purity of clustering results. At the algorithm's core are two directed $K$-nearest neighbour ($K$-NN) sparse graphs which direct the clustering of the algorithm. As with every $K$-NN graph-based algorithm, it requires the user to specify a number of fixed input parameters, most importantly the $K$ connectivity parameter. In this section we will describe how the base RepStream algorithm works and its input parameters, which are vital to the extensions that we present in Chapters 4 and 5.

RepStream takes as its input a stream of $d$-dimensional real-valued data points, where each dimension of the data point represents a numerical attribute. These data points are time-ordered and arrive one by one, incrementally. The two $K$-NN directed sparse graphs are constructed using the data points as the vertices, and are referred to as the point-level and the representative-level graphs, as shown in Figure 2.1.

### 2.3.1 Point Level Sparse Graph

The **point-level graph** is a $K$-NN directed sparse graph in which each vertex of the graph corresponds to one of the data points from the input data stream that is currently kept in memory. This graph describes the basic relationship between data points themselves. The number of vertices is limited by the memory capacity. An edge between two vertices represents a one-directional connection between the two corresponding data points.

Provided there are at least $K+1$ vertices in the sparse graph, each vertex in the graph has $K$ *outgoing* edges to nearby vertices, creating a one-way connection between the vertex and its $K$ closest neighbours with respect to the distance metric used. When a new vertex is processed by RepStream the data point is added to the point-level $K$-NN sparse graph - outgoing edges are created, linking to the $K$ nearest other vertices, and other nearby vertices also have their nearest neighbours readjusted if the new vertex is closer than its existing nearest neighbours.

The edges between vertices are directed, in that each edge has an explicit start and end point, and are not bi-directional, however it is possible for vertices to have edges which point to each other. If two vertices $v_1$ and $v_2$ have outgoing edges such that $v_1$ has an outgoing edge connecting to $v_2$ and $v_2$ has an outgoing edge connecting to $v_1$ then they are said to be *reciprocally connected*. An example of a reciprocal connection is shown in Figure 2.4 in which vertices $R_1$ and $R_2$ each have an outgoing edge connecting to the other.

While a vertex has a number of outgoing edges equal to the $K$ parameter, the number of incoming edges pointing to a vertex can be as few as zero, and also can be greater than $K$. Because of the nature of $K$-nearest neighbour graphs it is also possible for a vertex to have zero edges pointing to it, even though the number of outgoing edges is $K$; this tends to happen when a vertex is relatively very far away from other vertices. Edges also have a length which is dependent on the distance metric used, and RepStream has several options for distance metrics, including the Manhattan distance, Euclidean distance, Euclidean-squared distance, and Mahalanobis distance. For our purposes in this paper we use the Euclidean distance, which is the standard and most intuitive distance measure.

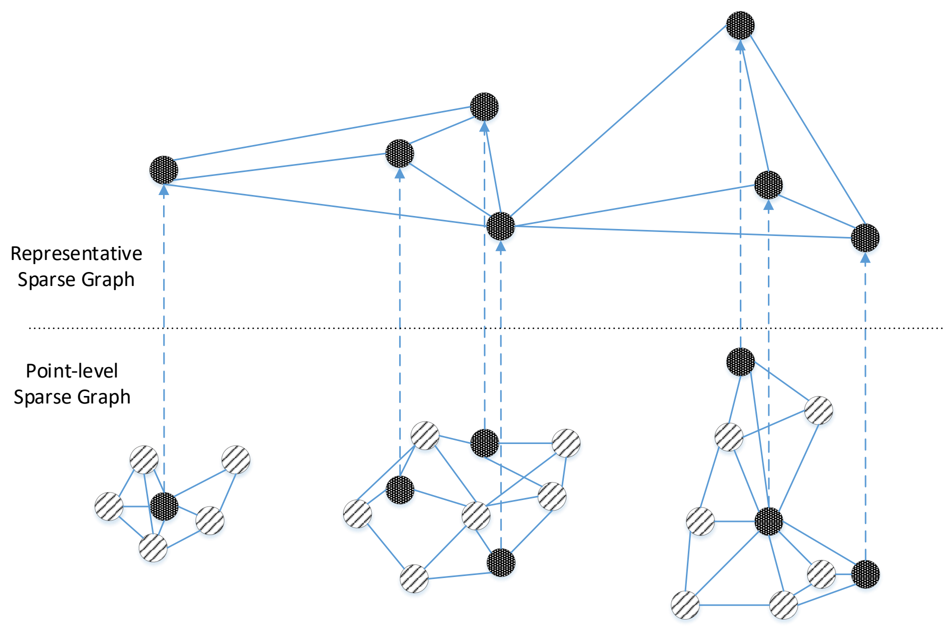


Figure 2.1: An illustration of the representative and point-level sparse graphs in the RepStream algorithm.

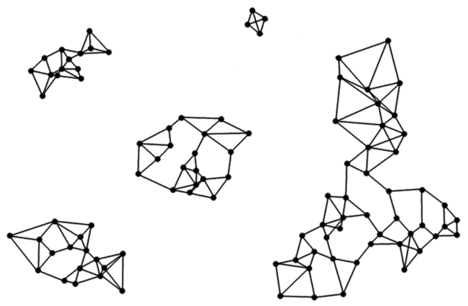
### 2.3.2 Representative Level Sparse Graph

The **representative-level graph** is the second sparse graph maintained in the RepStream algorithm. The representative-level graph is also a $K$-NN graph but its vertices consist of only *representative points*, which are a subset of data points currently kept in memory. A vertex becomes part of the representative-level sparse graph when it becomes a representative point. RepStream must maintain both of these graphs and update them as new data points are added to the stream which are added to the point-level graph and representative-level graphs as applicable.

Figure 2.1 shows the relation between the two different levels of sparse graphs in RepStream, and how a subset of the points in the point-level graph are used to construct a separate graph, which is later used in the clustering process of RepStream.

Representative points are points which, as their name suggests, represent points around them. They determine the cluster membership of the points they represent, and their position and connectivity to other representatives determines how clustering decisions are made. Figure 2.1 shows the relation between the point-level sparse graph and the representative-level sparse graph, in which a subset of data points are used to construct a second sparse graph on which clustering is performed.

Vertices become representative points when they are inserted into the graph and


(a) The position of representative points and the connectivity on the point-level graph in a hypothetical example in RepStream.

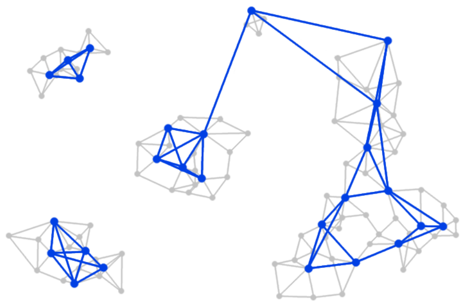

(b) The position of representative points and the connectivity on the representative-level graph in a hypothetical example in RepStream.

Figure 2.2: Representative compared to point-level sparse graphs.

have no reciprocal connection to an existing representative point. This method of selection allows the representative vertices to be more or less evenly spread through the data space. In this manner representative points are added incrementally as vertices are added to the point-level nearest neighbour graph when no neighbour representative is available.

To limit the amount of space used by RepStream the user may specify a limit to the number of data points kept in memory. This is done by indicating a maximum number of data points to be maintained in the point-level sparse graph. When a new vertex is added to the point-level graph, the oldest vertex which is *not* a representative will be removed from the sparse graph, and nearby neighbours will have their outgoing edges rearranged to maintain the $K$-nearest neighbour structure. Representatives vertices are not deleted in the same first in, first out way, instead, they are kept in a repository. Representatives are added to the repository as they are created, until the repository is full, once full the least useful representative is removed each time a new one must be added. This usefulness is determined by computing a *representative usefulness* value, which is dependent on the age of the representative, and the number of vertices that have linked to that representative.

Figure 2.2(b) shows an example of the representative points and the representative-level $K$-NN graph in a hypothetical with the corresponding point level graph in lighter grey. Representative points are added incrementally as the stream progresses, this means that as vertices are added to the point-level graph new representative points may be created, and so the representative level graph must be updated incrementally as well.

### 2.3.3 Clustering With RepStream

Forming individual clusters in RepStream is done at the representative level sparse graph, but using additional density information from the point level. This forms a combination graph and density based clustering method. Clusters are defined as groups of representative points which are both reciprocally connected, as we describe above, at the representative level sparse graph, and which are also mutually density related. Two representative points, and the points that they represent, will belong to the same cluster if both these criteria are met.

Because of its combination graph and density based approach, users do not need to specify a number of clusters to be found, or assume that clusters must be hyper-spherical. RepStream is able to determine the number of clusters based on its internal graph model, and can find arbitrarily shaped clusters.

As we mention previously, there are two criteria for merging two representative vertices into the same cluster, those are:

- Reciprocal connectivity at the representative-level
- Density relation

As we note in Section 2.3.1 reciprocal connections occur when a vertex $v_i$ has an outgoing edge to another vertex $v_j$, and the vertex $v_j$ has its own outgoing edge to $v_i$. In this case the vertices $v_i$ and $v_j$ are said to be reciprocally connected. In RepStream reciprocal connections at the representative-level are used for clustering decisions.

Density relation is a concept in RepStream which is determined by the relative distance of nearby points. Two representative points are said to be density related if they are within each other's density radius. The density radius for a vertex is calculated by taking the mean distance of the vertex's outgoing edges at the point level, multiplied by the $\alpha$ scaling factor. Two vertices must be within each other's density radius for them to be density related.

**Definition 2.3.1** *A representative vertex $v_i$ is said to be density related to a vertex $v_j$ if the following condition is met:*

$$Dist(v_i, v_j) < RelDens(v_i) \times \alpha \tag{2.1}$$

where $Dist(v_i, v_j)$ is the distance between the two vertices, $RelDens(v_i)$ is the relative density of the vertex $v_i$, which is defined as the average distance from $v_i$ to its $K$ nearest neighbours on the point-level sparse-graph, and $\alpha$ is user-set parameter for tuning the density relation. Figure 2.3 shows an example the density relation radius $\alpha \times AvgDist$ for the vertex.

Figure 2.4 shows an example of two representative vertices $R_1$ and $R_2$ which are reciprocally connected by edges $E_1$ and $E_2$. The two vertices are also within each other's density relation radius, denoted by the radii around the vertices. The vertex $R_1$ is within the density radius of $R_2$, and $R_2$ is inside the density radius of $R_1$. In this case since they are both reciprocally connected and density related they would belong to the same cluster, and all nearby point-level vertices they represent would belong to the cluster as well. On the other hand, Figure 2.5 shows an example of two representative vertices which are reciprocally connected, but not density related, and therefore would not be merged into the same cluster.

Cluster formation works in a transitive way, such that if vertex $v_1$ is in a cluster with $v_2$, and $v_2$ is also in a cluster with $v_3$, then both $v_1$ and $v_3$ are in the same cluster. If a representative point is not reciprocally connected and density related to any other representative points then it will belong to its own cluster. All vertices which are not representative points will belong to the same cluster as their nearest representative vertex, thus all points in the data set will belong to an existing cluster at all times. Figure 2.6 shows how representative points can group together to form clusters when they are reciprocally connected and density related.

As vertices and edges are added and removed from both the point and representative level sparse graphs over time, these clusters can change. Two vertices can lose their reciprocal connections as the $K$ neighbourhood shifts, or can lose their density relation if the relative density of a vertex changes. When this happens clusters can split if there is no other existing transitive relation. Similarly, clusters can merge together when vertices gain reciprocal connections or gain mutual density relation.

### 2.3.4 Algorithm

To demonstrate how RepStream processes new data points, Algorithm 1 shows the high level steps that a new vertex goes through. A new vertex $v_{new}$ is added to the point-level sparse graph, having edges created between neighbouring vertices using the $createLink(v_i, v_j)$ function. The function $NN(v_i)$ denotes the set of existing neighbours for the vertex $v_i$, whilst $FarEdge(v_i)$ will return the edge connecting to its farthest

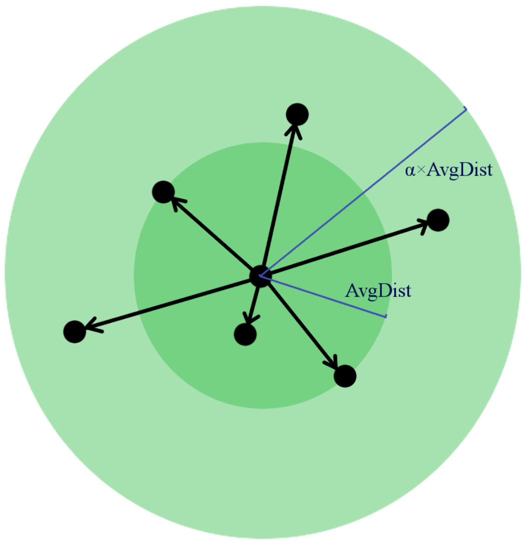


Figure 2.3: The density relation radius of a representative point is equal to $\alpha \times AvgDist$ where $AvgDist$ is the average distance to its $K$-nearest neighbours.

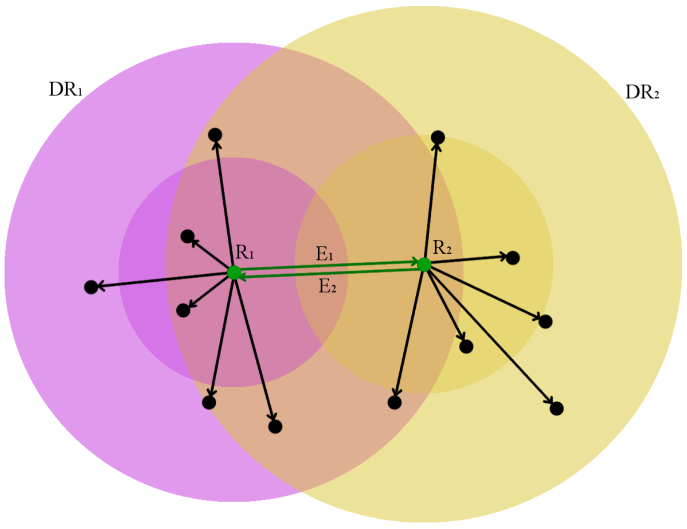


Figure 2.4: An example of two representative points which are both reciprocally connected, and density related.

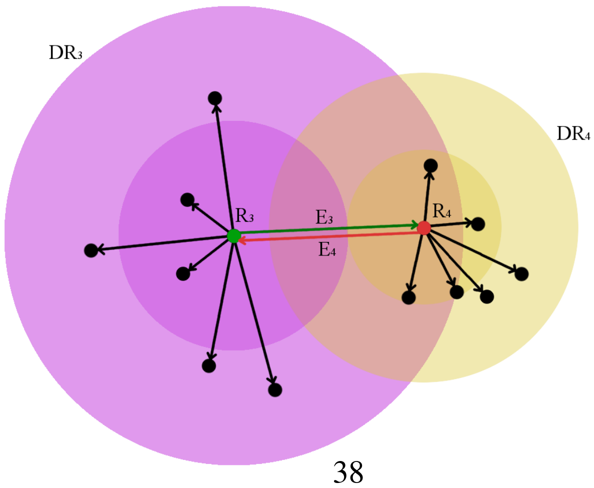


Figure 2.5: An example of two representative points which are reciprocally connected, but not density related.

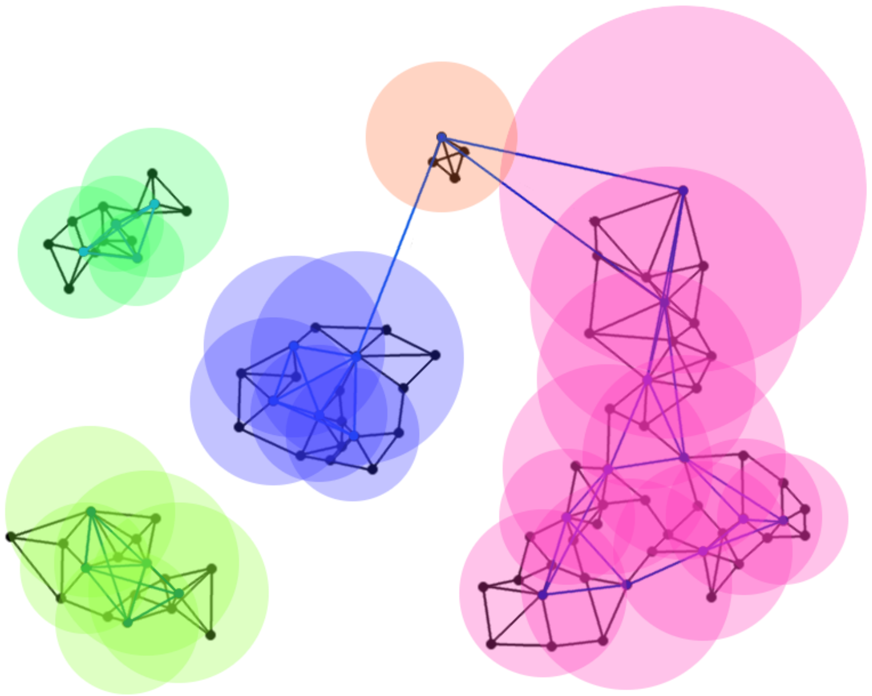


Figure 2.6: Density relation radii for the representative points in the hypothetical example in RepStream. The distance to the neighbours at the point level determines the density relation radius for the representative level clustering.

neighbour in its $K$ neighbourhood, which is determined by the distance between two vertices, denoted by $dist(v_i, v_j)$. Adding a vertex to the sparse graph can cause the $K$ neighbourhoods of existing vertices to shift, which may cause edges of existing vertices to be replaced. If a vertex being linked into the sparse graph has no existing reciprocal connection to a representative vertex, then it itself becomes a representative, which is shown as $makeRepresentative(v_{new})$.

Linking a new representative vertex into the representative level sparse graph follows a similar process, shown in Algorithm 2. The vertex has edges created to nearby representatives in its local $K$ neighbourhood, using the representative level function $createLinkR(r_i, r_j)$. The functions $NNr(r_i)$ and $FarEdgeR(r_i)$ return, respectively, the set of $K$ nearest representative vertices, and the farthest existing representative in the $K$ neighbourhood. Once the new representative is connected to its $K$ nearest representative neighbours, it may merge with existing clusters if it shares a reciprocal connection to a neighbour, whilst also being density related. As we mention before, this is a transitive property, so it's possible for groups of representatives to be merged into a large cluster.

**Algorithm 1** Algorithm for RepStream Algorithm

```
$FUNCTION : LinkIntoGraphSG$
$INPUT : v_{new}, neighbours$ {new vertex and neighbours}
for all $v_j$ in $NN(v_{new})$ do
  $createLink(v_j, v_{new})$
  if $|NN(v_j)| < k$ or $dist(v_{new}, FarEdge(v_j)) < dist(v_j, FarEdge(v_j))$ then
    $createLink(v_j, v_{new})$
    if $|NN(v_j)| > k$ then
      Remove edge to farthest neighbour of $v_j$
      update local density of $v_j$
    end if
  end if
end for
for all vertex $v_j$ which has a new reciprocal link do
  if $v_j$ is a representative vertex then
    Update representative reinforcement of $v_j$
    Delete and unlink least useful representative if repository is full
  end if
end for
if $v_{new}$ not reciprocally connected to a representative then
  $makeRepresentative(v_{new})$
end if
if $v_j$ not reciprocally connected to a representative then
  $makeRepresentative(v_{new})$
else
  assign $v_{new}$ to nearest representative
end if
for all vertex $v_j$ with removed edges do
  if $v_j$ not reciprocally connected to a representative then
    $makeRepresentative(v_j)$
  end if
end for
```

**Algorithm 2** RepStream representative level link-in algorithm

```
$FUNCTION : LinkIntoGraphRSG$
$INPUT : r_{new}, neighbours$ {new representative vertex and neighbours}
for all $r_j$ in $NNr(r_{new})$ do
   $createLinkR(r_j, r_{new})$
   if $|NNr(r_j)| < k$ or $dist(r_{new}, FarEdgeR(r_j)) < dist(r_j, FarEdgeR(r_j))$ then
      $createLinkR(r_j, r_{new})$
      if $|NNr(r_j)| > k$ then
         Remove edge to farthest neighbour of $r_j$
      end if
      if $densityRelated(r_{new}, r_j)$ then
         Merge clusters of $r_{new}$ and $r_j$ if not already in same cluster
      end if
   end if
end for
for each vertex $r_j$ with removed edges do
   Check if cluster of $r_j$ must be split
end for
```

### 2.3.5 Parameters In RepStream

RepStream's primary parameter is the graph connectivity parameter $K$. The $K$ value in this case is incredibly vital to the RepStream algorithm, because it determines the level of connectivity in both the point-level and representative-level sparse graphs. This affects the connectivity of data points in memory, and is the primary parameter in determining the final clustering output. A higher $K$ value causes the data points to be more connected, resulting in fewer but larger clusters, while a lower $K$ value causes clusters to be more fragmented due to lower graph connectivity. Because of this, using a value of $K$ appropriate for the data set is important for achieving high quality clustering results, and selecting $K$ appropriately is vital for achieving satisfactory clustering results.

There are other parameter values used by RepStream as well, specifically the $\lambda$ decay factor, the $\alpha$ scaling factor . The $\lambda$ decay factor determines how quickly representative points in RepStream are removed over time. The $\alpha$ scaling factor is used in determining when nearby representative points will be placed into the same cluster. The higher the $\alpha$ value, the farther away two mutually connected vertices may be from each other before they are merged, as we explained previously. The $\lambda$ and $\alpha$ values are important to RepStream's operation, however the algorithm is less sensitive to their values, and thus they are easier to set. Specifically, the density component of RepStream is controlled by the $\alpha$ value. Since representative-level neighbours are expected to be farther away than point-level neighbours, values of $\alpha < 1$ will prevent representative points from being clustered together. As such $\alpha$ must be greater than 1, but not so great as to group representatives together too aggressively. Experimentally, values between 1 and 3 tend to produce the best overall results.

# Chapter 3

# Change Detection in Data Streams[1]

Our first task is to demonstrate that concept drift is detectable and can be identified algorithmically soon after the changes occur. This is an application of the change detection field on arbitrary streams of multi-dimensional data. In this chapter we present a method for identifying change points in data streams for the purpose of aiding stream clustering algorithms.

## 3.1 Overview of Stream Change Detection

As we have previously mentioned in Chapter 1, concept drift is a major challenge in stream data mining. The unpredictable evolution of concepts within the stream results in problems for both supervised and unsupervised methods. Supervised methods, such as stream classification, can suffer from training data becoming outdated, while unsupervised methods, like stream clustering, may have unreliable performance due to static initial parameters (Gama et al., 2014).

Many older clustering algorithms rely on the assumption that the dataset they are to analyse is static, as is the case with batch datasets, for example the classic DBScan (Ester et al., 1996) and K-Means (MacQueen et al., 1967) algorithms. Applied to a data stream, algorithms such as these suffer from the problem of conflating old and new data, possibly never reaching a useful clustering solution.

This problem has led to the development of stream clustering algorithms, which

[1]*Portions of this chapter are copyright ©2015, Australian Computer Society, Inc. Such portions appeared at the Thirteenth Australasian Data Mining Conference, Sydney, Australia. Conferences in Research and Practice in Information Technology, Vol. 168. Md Zahidul Islam, Ling Chen, Kok-Leong Ong, Yanchang Zhao, Richi Nayak,Paul Kennedy, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.*

are specifically designed to handle streaming data. One common method of handling change in data streams is to use a sliding window (Silva et al., 2013), where only the newest set of points are used to make decisions. Regardless of how exactly this is achieved a successful stream clustering algorithm must be able to keep up with changes in the data, as not doing so risks sub-optimal performance.

While there are many current algorithms which are specifically designed to handle stream data, including methods that allow them to cope with changes (Bhatnagar et al., 2014; Chen & Tu, 2007; Amini et al., 2014; Aggarwal et al., 2004), they are limited by their reliance on initial static operating parameters and therefore vulnerable to problems associated with concept drift. Identifying when these changes occur in a stream is a critical component to dealing with the evolution of data streams. Our first task is to show that concept drift can be detected by analysing the data structures created and stored by a data mining algorithm running on the stream. Being able to detect the points at which stream evolution occurs allows us to design algorithms which are able to make more informed decisions that allow them to adapt to the changes in the stream.

In addition to the necessity of keeping up to date with new instances of the data, the static parameters of many algorithms means that the algorithms may degrade in performance at certain times during the stream. Almost all algorithms are given initial input parameters which determine how they operate on the data. Examples include D-Stream (Chen & Tu, 2007) which uses $C_m$ and $C_l$ as density thresholds, ExCC (Bhatnagar et al., 2014) which requires a grid-granularity parameter, or SWClustering (Zhou et al., 2008) which uses $\beta$ a radius threshold and $N$ to define the number of sub-cluster summaries it stores.

The parameters are typically set at the start of a clustering operation, and remain unchanged as the stream progresses. This can pose problems since, even assuming the parameters are set optimally for the start of the stream, there is no guarantee that the parameters will produce good results at a later point in the stream. Thus, not only should the algorithm be able to keep up to date with newer data, but it should also be able to adapt how it operates to match potential unknown changes in the data stream. These in combination should lead to algorithms which perform better in the face of dynamic streaming data. The fact that many existing algorithms rely on this sort of static critical parameter is one of the major challenges in stream clustering (Silva et al., 2013).

In this chapter we propose a method of data stream change detection by using structural features of a $K$-nearest neighbour graph structure in the clustering algorithm Rep-

Stream. We have selected RepStream as the base algorithm due to its directed sparse-graph data structure, and because it has been shown to effectively cluster streams with arbitrarily shaped distributions. These methods could, however, be generalised for use in other nearest-neighbour directed sparse graph stream mining algorithms. We use RepStream's internal graph structures because of its efficient use of representative points to reduce computational complexity when dealing with graphs containing many data points.

RepStream (Lühr & Lazarescu, 2009) is a stream clustering algorithm using a combination of density and *K*-nearest neighbour graph-based approaches. Of specific interest to us is the so called '*K* value', which is the parameter concerned with the degree of connectivity in the graph-based data point representation. Varying the *K* value has a dramatic effect on the clustering performance of RepStream and, as with most algorithms, selecting this parameter poorly can lead to undesirable clustering results. Whilst a single *K* value works for batch datasets or streams where the distribution changes little over time, a data stream which evolves significantly might require different *K* values at different times to achieve optimal results. Since it is not safe to assume that a stream will remain stable over time we should be looking at methods to deal with evolving distributions. Thus, it is of interest to us to determine the best *K* value to use at different points in time. In essence we propose that *K*-nearest neighbour algorithms suffer because of their static *K* values. This is the fundamental parameter of such algorithms, and by varying them in response to stream changes they become more suited to streaming contexts.

The focus of our task is determining *when* we are likely to need to change *K*. This occurs when the distribution of the data changes significantly, so knowing when significant change occurs within the dataset is extremely important. Therefore we want to determine when the distribution changes in a way which affects RepStream's clustering performance.

In this chapter we propose a method to detect structural changes in a data stream. We achieve this by examining features extracted from the *K*-nearest neighbour graph structure within RepStream. A typical change detection approach is done by calculating a probability distribution for a certain window in a stream, then comparing its distance to a later reference window (Kifer et al., 2004). Our approach differs in that we first extract higher-level features from a *K*-nearest neighbour graph before performing change detection. We will demonstrate that these features mirror the structure of the underlying data stream, and can be used to detect when changes occur in the stream.

### 3.1.1 Contributions

The contributions of this chapter are:

- A definition of structural data stream changes that takes into account higher-level computed features and statistics which are also used in stream clustering.
- Extraction of structural features in RepStream and an evaluation of their effectiveness for locating changes in data streams.
- A method by which we can locate changes in the data stream by examining structural features of the $K$-nearest neighbour clustering algorithm, RepStream.
- An examination of, and evaluation on the KDD 99' Intrusion Detection dataset, which is commonly used as a benchmark in stream mining contexts.

The chapter is organised as follows. In Section 3.2, we define basic terms used in the paper, as well as definitions of the features which we extract for analysis. In Section 3.3, we present our method for defining change points, as well as our change detection algorithm. Section 3.4 contains a detailed analysis of the KDD Cup 99' dataset, as well as a thorough evaluation of our method and a comparison method on the dataset. In Section 3.5, we discuss the results, and the strengths and weaknesses of the algorithm. Finally in Section 3.6 we summarise the chapter.

## 3.2 Definitions

First we will define the terms and methodologies we use. We describe data distribution, cluster assignments, and how we define the changes that we are detecting. We also define and describe the features we extract from the $K$NN graphs, which we use for our change detection method.

### 3.2.1 Basic Definitions

In this chapter we restrict our attention to change detection in data streams. A data stream $\mathscr{S}$ is an unbounded series of time-ordered data points $\mathscr{S} = \{X_1, X_2 \ldots X_t \ldots\}$. Each data point $X_i$ is a vector with a number of attributes $X_i = \{x_{i,1} \ldots x_{i,d}\}$, where $d$ is the dimensionality of the vector. Our work concentrates on real-valued data clustering, so the attributes in the vector are real valued, and can be represented by a point in

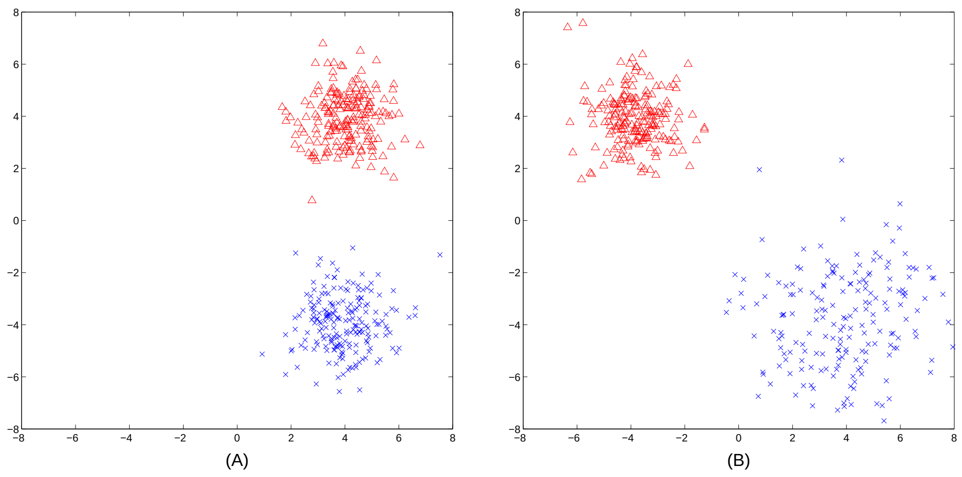


Figure 3.1: Example of distribution change in a stream. Distribution A shows two classes of the same density at different spacial locations. At a later time in the data stream, distribution B, the red cluster has shifted its position, whilst the blue cluster has increased its size, and decreased its density.

$d$-dimensional space.

These data points are sampled at each time step from a distribution. In a data stream we have a set of distributions over time $\mathscr{D} = \{D_1, D_2 \dots D_t \dots\}$ such that each data point $X_i \in \mathscr{X}$ is selected from a distribution $D_i$. The distribution may either change or stay the same from one time step to the next. We define a change in the data stream as being when the distribution at one time step is no longer the same as the distribution at the previous time step, while there is no change when $D_i$ is equal to $D_{i-1}$. It is possible for the distribution to remain unchanged for any amount of data points, or even the whole stream. Figure 3.1 shows a simple example of such a change in the distribution of a stream – at some time $T_1$, (A), the data points are sampled from a distribution with two classes, red and blue, which are the same size and density. At a later time $T_2$, (B), the red class has changed its position in one of the data dimensions, effectively moving its location in the multidimensional space, while the blue class has increased in size and decreased in density.

To define clustering, at each time step data points will be split into a grouping of clusters $\mathscr{C} = \{C_1, C_2 ... C_t\}$ such that each cluster grouping $C_i$ represents the configuration of clusters at time $i$. A cluster configuration $C_i$ contains between one and $n$ clusters $C_i = \{c_1, c_2 \dots c_n\}$ where $n$ is at most the number of data points maintained in memory. If $\mathscr{X}_t$

is the set of all data points currently in memory, then each data point is labelled as part of a cluster $\forall X_i \in \mathscr{X}_t, \exists c_j \in C_t$ such that $X_i$ is labelled as a member of the cluster $c_j$.

The set $G_t = \{g_{t,1} \ldots g_{t,n}\}$ represents the cluster assignments which are theoretically ideal, this is defined as $\forall X_i \in \mathscr{X}_t\ \exists g_{t,j} \in G_t$ such that $X_i \in g_{t,j}$, where $g_{t,j}$ is the grouping that corresponds to a class within the distribution $D_t$. Thus the groupings represented in $g_{t,j}$ match the distribution at that time.

The algorithm we concentrate on in this paper is RepStream (Lühr & Lazarescu, 2009), which uses a $K$-nearest neighbour graph representation of the data points. RepStream uses a directed sparse graph, which we can denote as $SG(V_t, E_t)$, where $V_t$ is the set of vertices at time $t$ and $V_t = \{v_{t,1}, v_{t,2} \ldots v_{t,n}\}$. Each vertex $v_{t,i}$ matches to a corresponding data point $\mathscr{X}$. The symbol $E_t$ represents a series of directed edges between vertexes $E_t = \{e_{t,1}, e_{t,2} \ldots e_{t,m}\}$ at time $t$. Each edge $e_{t,i}$ is an ordered pair that represents the start and end points of an edge in the sparse graph, $e_{t,i} = \{u, v\}$ where $u, v \in V$. The number of edges $m$ is determined by the $K$ value, as each vertex has $K$ outgoing edges (whenever the number of vertexes is greater than $K$), so $|E_t|$ will never be greater than $k \times |V_t|$.

### 3.2.2 Feature Definitions

We extract five features from the $K$-nearest neighbour structure of RepStream while it is performing its clustering. These features, defined below, reflect the structure of the $K$-nearest neighbour sparse graph maintained by RepStream. They signify elements of the incoming data which may be affected by changes in the underling concepts. Our intuition is that significant changes in these high-level computed features would reflect significant evolution in the data.

**Cluster Count** Cluster count is simply the number of clusters currently being reported by RepStream at the current point in time. Cluster count is defined as follows:

$$ClusterCount = |C_t|. \tag{3.1}$$

Extracting the cluster count is computationally efficient. Since RepStream produces a clustering result for every data point inserted into the algorithm, this feature is

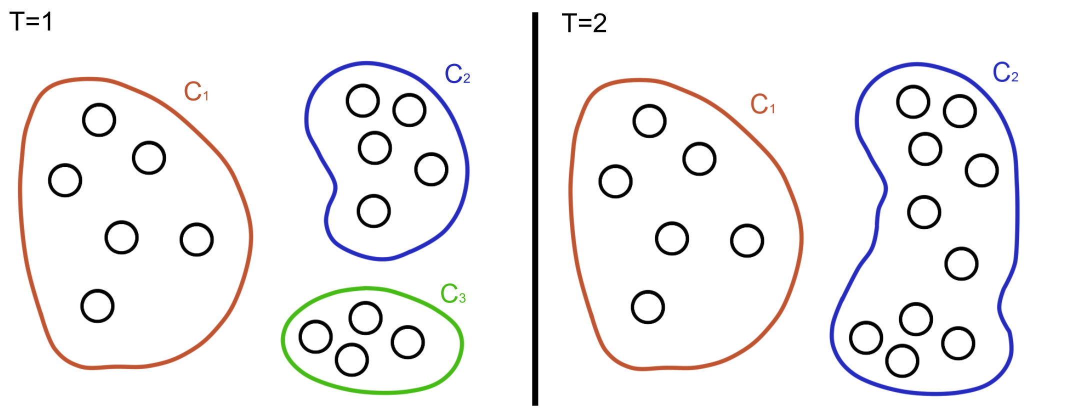


Figure 3.2: Cluster Count. In this example at $T = 1$ there are 3 clusters. The addition of an extra point at $T = 2$ results in two clusters merging, reducing the cluster count to 2.

simply the number of clusters that RepStream locates at each time step. However, it is important to note that this feature is dependent on the $K$ value used. A higher $K$ value will typically result in a lower number of clusters found by the algorithm, due to the higher connectivity of the $K$-nearest neighbour graph. We found that lower values of $K$ – approximately $K = 10$ – are more sensitive to changes in the data stream. This is possibly because higher $K$ values are likely to have many strong connections, 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 edge change count represents the number of edges which are created and destroyed over time on the representative layer. The representative layer is a second sparse graph which is made of a subset of all points in memory. RepStream uses a first-in-first-out queue for vertexes in the graph. $V_t = \{v_{t,1} \ldots v_{t,m}\}$ where $m$ is some memory limit defined by the user. If the data points existing in RepStream's sparse graph at time $t$ are $V_t = \{v_t, v_{t+1} \ldots v_{t+m}\}$ then at time $t+1$ the data points in the sparse graph will be $V_{t+1} = \{v_{t+1}, v_{t+2} \ldots v_{t+m+1}\}$.

The edge change count necessarily changes the membership of the edges $E$ of the sparse graph as well, because whenever a new representative is added a previous vertex is removed, which causes all outgoing edges and incoming edges pointing to that representative vertex to be removed. The minimum number of edges that are removed from the set of edges $E$ when a single point is added is equal to $K$ (assuming $|V| > K$).

Similarly, at least $K$ new edges must be added to $E$ due to the new vertex requiring $K$ new edges. More edges may be added or removed if the removed representative vertex was a nearest neighbour of other vertices, or if the new vertex becomes a nearest neighbour of existing vertices. The definition of edge change count is:

$$EdgeChangeCount = AE(E_i, E_j) + RE(E_i, E_j). \tag{3.2}$$

In the above equation $AE(E_i, E_j)$ is a function that returns the number of edges that have been added to the sparse graph. They are present in $E_j$ but not in $E_i$. The symbol $E_i$ is the set of edges in the sparse graph at time $i$ as defined above. Thus,

$$AE(E_i, E_j) = |E_j| - |E_i \cap E_j|. \tag{3.3}$$

Similarly, $RE(E_i, E_j)$ is a function that returns the number of edges which have been removed from the sparse graph in $E_j$:

$$RE(E_i, E_j) = |E_i| - |E_i \cap E_j|. \tag{3.4}$$

The edge change count shows the number of edges created and removed at the representative layer of RepStream 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** Cluster merges-and-splits counts the number of times clusters have joined together or split into two or more separate clusters since the last time it was measured. If $C_i$ is the arrangement of clusters at time $i$, and $C_j$ is the arrangement of clusters at a later time $j$. Then the set of new clusters is defined as:

$$C_{new} = C_j \setminus [C_i \cap C_j]. \tag{3.5}$$

The new clusters which were split from existing clusters are given by:

$$C_{splits} = \text{the set of } c_x \subset C_{new}, \text{ where } c_x \neq \{v_j\}. \tag{3.6}$$

Similarly, the set of clusters that have been removed is:

$$C_{removed} = C_i \setminus [C_i \cap C_j]. \tag{3.7}$$

Thus, the set of clusters which have merged into existing clusters is:

$$C_{merged} = c_x \subset C_{removed}, \text{ where } \exists c_x \subset C_{removed}, c_y \subset C_j | c_x \subset c_y. \tag{3.8}$$

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 KNN 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 data points into a stable configuration.

**Edge-Length Variation** The edge-length variation of a vertex is the total amount of variance there is amongst the lengths of all edges in the dataset.

It is defined as:

$$EV(v_{t,i}) = \sqrt{\frac{\sum\limits_{j=1}^{k} (|e_{i,j}| - \bar{|e_i|})^2}{k}}, \tag{3.9}$$

where $e_{i,j}$ is the $j^{th}$ outgoing edge of vertex $v_{t,i}$.

Thus, the average edge-length variation for the whole window would be given by summing and taking the average of the edge-length variation for every vertex $v_i \in V$ in $SG(V,E)$:

$$AEV(V_t) = \frac{\sum\limits_{i=1}^{n} EV(v_{t,i})}{n}. \tag{3.10}$$

Edge-length variation is computed 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 currently in memory. The standard deviations are then added together and divided by the total number of points in the $K$-nearest neighbour 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 $K$-nearest neighbour 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** The History count represents the cumulative number of times that points returned to clusters they were previously members of, across the whole dataset.

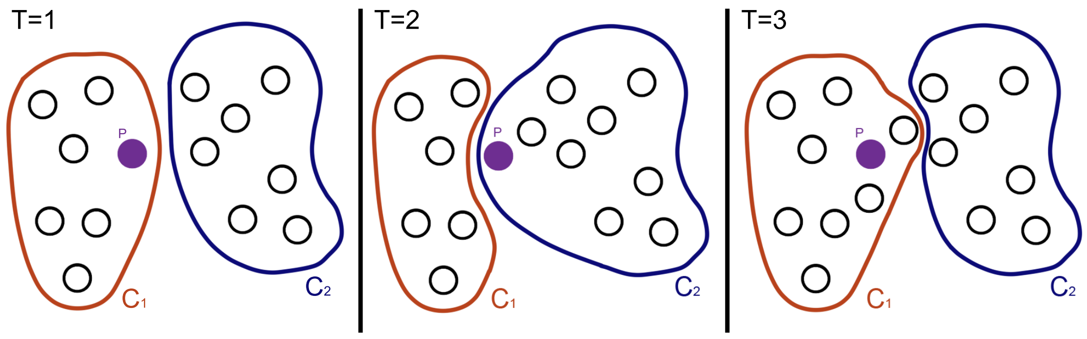


Figure 3.3: history count. Point $P_1$ begins in cluster $C_1$ initially, then at time $T = 2$ the addition of a new data point causes a change in the membership to cluster $C_2$. Later at $T = 3$ it returns to the cluster it was previously in.

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.

Let the history of a vertex $v_i$ be $H_i = \{h_{i,1}...h_{i,p}\}$ where each history instance $h_{i,j}$ represents the cluster that $v_i$ was assigned to at time $j$, and $p$ can be between 1 and the age of the vertex in time-steps. The history count would be the total number of times that the vertex returned to a cluster that it was previously assigned to:

$$HC(v_i) = |H_i| - |unique(H_i)|, \tag{3.11}$$

where $unique(H_i)$ is a function that returns the set of history instances that are unique within the history of the vertex. As an example, let some vertex $v_i$ have a history $H_i = \{c_1, c_2, c_3, c_1, c_4, c_2, c_1\}$, where each element of $H_i$ is the identifier of some cluster. The history count of the vertex would be $HC(v_i) = |\{c_1, c_2, c_1\}| = 3$. This reflects the number of times that the vertex has re-joined a cluster it has previously been a part of.

## 3.3 Methodology

We begin by extracting a number of geometric and structural features from the nearest neighbour graph-based data representation of RepStream. Using these we turn the difficult process of detecting arbitrarily dimensional distribution change into a time series change point problem.

We then use our time series change point detection method on these features to determine when changes occur.

Finally, we use a voting system to determine when change has occurred according to analyses of these features.

### 3.3.1 Feature Extraction

We begin by extracting each of the features described in the previous section. For our purposes RepStream was run at a fixed limit of 1000 points stored in memory at a time, in a first-in-first-out queue. This was selected due to such a memory limitation producing desirable results in the original paper (Lühr & Lazarescu, 2009).

Our features are extracted every 100 data points, which is 10% of the memory horizon. Though it is possible for cluster arrangements to change rapidly with the addition of a single point, it is not realistic for any algorithm to be able to reliably detect change with only a single data point (or even a handful). This is particularly the case in a dataset with noise. A sampling frequency equal to 10% of the memory limit was selected as it is a significant portion of the amount of data points in memory without being evaluated too often and driving up complexity. Thus, our algorithm evaluates whether change occurs every 100 data points, and each evaluation represents the activity over that period.

Both the memory limit of RepStream and the frequency of the feature extraction can, of course, be adjusted at will for different purposes. However this is what they have been set at for our experiments.

### 3.3.2 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 point detection algorithm. The algorithm is shown as Algorithm

3. The inputs are listed and given as $(feature, M, H, \lambda)$. The input $feature$ represents the given feature as a time series. The $M$ parameter 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 a moving average. Finally, $\lambda$ is a parameter that determines how quickly the algorithm updates to match newer observations, if it is set to zero then the algorithm will not adapt to slow changes, meaning that even the most gradual changes will be detected over time. The algorithm returns *changes* which contains a list of time indexes where changes have occurred in the time series $feature$.

In the following algorithm the input $feature$ is the feature series, $M$ is the sensitivity parameter, $H$ is the number of previous points to calculate moving mean/deviation using, and $\lambda$ determines how fast the mean and deviation adjust to new data.

**Algorithm 3** Detection Method

```
1: $FUNCTION : ChangeDetect$
2: $INPUT : X, M, H, \lambda$
3: $OUTPUT : Changes$ {a list of indexes where changes have been detected}
4: $changes = \emptyset$
5: $Y = \text{mean}(feature(1 : H))$
6: $Z = \text{standardDeviation}(feature(1 : H))$
7: for i = M : size(feature) do
8:     $\bar{X} = \text{mean}(feature(i - H : i))$
9:     $\sigma = \text{standardDeviation}(feature(i - H : i))$
10:    if abs$(\bar{X} - Y) > M \times Z$ then
11:        $changes = changes + \{i\}$
12:        $Y = \bar{X}$
13:        $Z = \sigma$
14:    else
15:        $Y = (1 - \lambda) \times Y + \lambda \times \bar{X}$
16:        $Z = (1 - \lambda) \times Y + \lambda \times s$
17:    end if
18: end for
```

The algorithm calculates $Y$, the mean over the first $H$ data points, and $Z$, the standard deviation over the first $H$ data points. These adjust over time with respect to $\lambda$ until a change occurs, at which point they are recalculated for the most recent $H$ data points. When the algorithm begins they are set to the average and standard deviation over the first $H$ points of the time series, and they are recalculated over the previous $H$ points whenever a change is detected. The variable $\bar{X}$ tracks the moving average of the input time series, and when $\bar{X}$ goes outside the range defined by $Y \pm H \times Z$, that is -

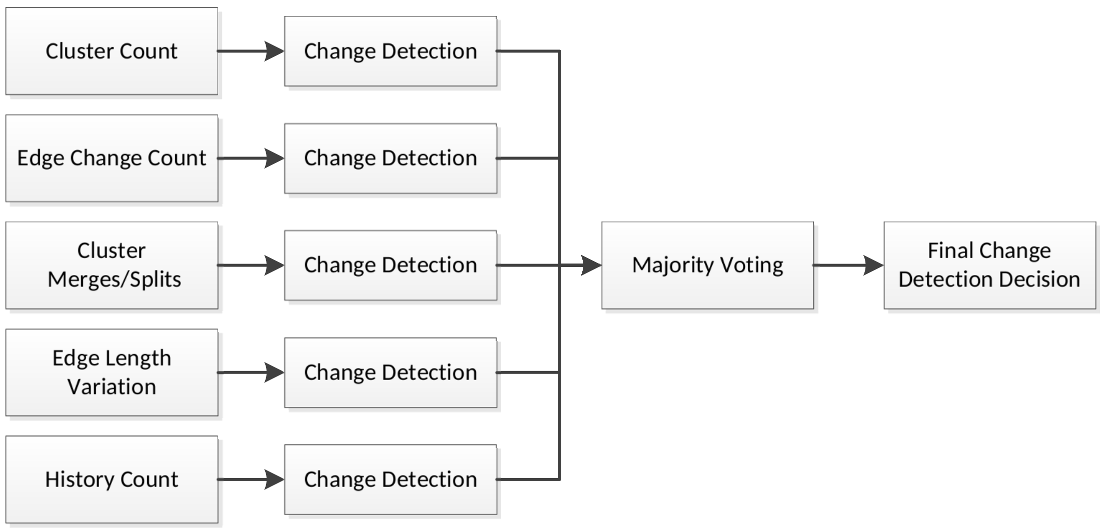


Figure 3.4: Flowchart showing feature extraction, detection on each feature, then majority voting to reach the final change detection decision.

$\bar{X} > (Y + H \times Z)$ or $\bar{X} < (Y - H \times Z)$ - then we consider a change to have occurred. If no change was detected during that time-step then $Y$ and $Z$ are shifted slightly towards the current moving average and standard deviation $\sigma$ by a fraction defined by $\lambda$.

The advantage of this algorithm is that it gives the flexibility of detecting rapid or arbitrarily slow changes depending on how $\lambda$ is set. A higher value of $\lambda$ means that changes need to be more sudden and dramatic to be detected, while a low $\lambda$ value will let the algorithm detect changes that occur slowly over large time periods.

Since there are five testing features, we use a system where the detection process run on these features must agree before a structural change is detected. Detections for at least $N$ features must agree that there has been a change within the last $T$ samples for the algorithm to detect a change. If $N$ is greater than half the number of features then it is simply majority voting. Figure 3.4 shows the process. Each feature is extracted, and our change detection is run on each individually, finally majority voting is used to decide whether a change is reported or not.

### 3.3.3 Defining Ground Truth Changes

For evaluation purposes we need to define where the stream distribution change points occur in the data stream, and then evaluate the outcomes of our algorithm based on these change points. The KDD Cup 99' dataset (Stolfo et al., 2000) provides a set of

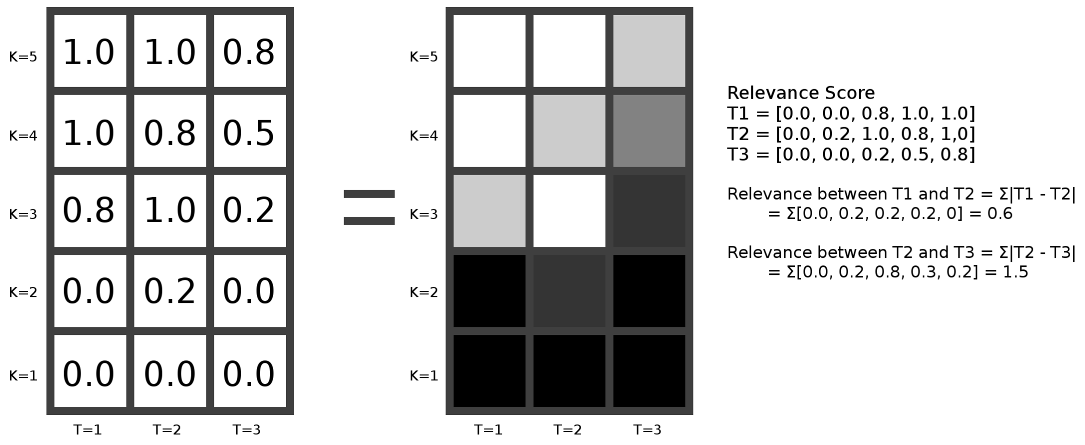

K=5
K=4
K=3
K=2
K=1
1.0 1.0 0.8
1.0 0.8 0.5
0.8 1.0 0.2
0.0 0.2 0.0
0.0 0.0 0.0
T=1
T=2
T=3
Relevance Score
T1 = [0.0, 0.0, 0.8, 1.0, 1.0]
T2 = [0.0, 0.2, 1.0, 0.8, 1,0]
T3 = [0.0, 0.0, 0.2, 0.5, 0.8]
Relevance between T1 and T2 = Σ|T1 - T2|
= Σ[0.0, 0.2, 0.2, 0.2, 0] = 0.6
Relevance between T2 and T3 = Σ|T2 - T3|
= Σ[0.0, 0.2, 0.8, 0.3, 0.2] = 1.5

Figure 3.5: Illustrative example of the F-Measure heatmap.

class labels for each data point, this allows us to define ground truth change locations for our evaluation. To define change points we split the dataset into blocks of 1000 points, and if the same set of classes are present from one window to the next then there is no change in the data set. If the set of classes from one window to the next is different then we define a change point within that window.

We weight change points according to a relevance score. This is because there are some changes, as previously described, where a class may only be present for a handful of data points before disappearing. Changes such as this may ultimately have little to no effect on the actual structure of a dataset. As such, we have developed a system to weight the change points in relation to how much it affects the clustering performance of RepStream. We do this by generating what we call a heatmap. This is a 2-dimensional table where each row represents the clustering performance of RepStream (in terms of F-Measure) on the data set at each time step at a given $K$ value. For this purpose we use the assumption that the clustering performance varies more when there is a more significant change in the data stream and less when there is a less significant change. As such, we weight change points proportionally to the total amount the F-Measure varies from one time step to the next across all $K$ values.

To illustrate this, Figure 3.5 shows two grids as an example. The grid on the left shows the F-measure value for each time-step for a range of $K$ values, on the right is a visualisation which shows the same values, presented as shaded cells, where an F-measure value of 0 is displayed as black, and a value of 1 is displayed as white. The relevance between a given time step to the next time step is defined as follows:

$$R(T_1, T_2) = |T_1 - T_2|. \tag{3.12}$$

However, because some changes may have a faster effect on the dataset, and some may act very slowly, we have defined the weight as being the variation over a certain amount of time. The total weight of a change point at time $T_i$ is defined like so:

$$Weight(T_i) = \max_{r=(1,2...range)} (R(T_{i-r}, T_{i+r})). \tag{3.13}$$

This gives us a range before and after the ground-truth change point, and allows us to judge how much of an effect the change has on the performance. We have set $range = 5$, which corresponds to 500 points before and after the change point – the size of the change point window – to allow for even the slowest possible changes to be detected. The algorithm selects the maximum change within this region, and weights the change point according to that.

Despite the fact that the ground truth changes are calculated only every 1000 points there are still a very large number of changes in the dataset. To handle this we have added a threshold $\theta$, such that any change point $CP_i$ where $|CP_i| < \theta$ is discarded. This produces a dataset with a smaller number of more significant change points, which more closely reflect the changes that we are looking for. We have set this threshold to be $\theta = 4$, where the maximum possible value for the weight is 26. This has reduced the number of change points from 89 to 43, cutting out the lowest weighted ones, many of which were clustered around higher weighted change points.

By weighting the ground truth change points as we do we filter out less dramatic changes, which have little to no effect on the dataset. Section 3.4.2 contains an analysis of the various classes in the dataset, as well as their comparative sizes. The change points which we weight more highly are the ones which cause a larger effect on the performance of the clustering algorithm, and which have a larger change in distribution. These changes are more important to detect and are likely to cause larger structural changes in the $K$-nearest neighbour structure of RepStream.

## 3.4 Experiments

### 3.4.1 Evaluation Measures

To evaluate the performance of our proposed method, we use the following common measures in the change detection literature (Gustafsson & Gustafsson, 2000; Basseville et al., 1993):

- **The False Alarm Rate**, which is the probability that the algorithm indicates a change point, but which no actual ground truth change point is present.
- **The Detection Rate**, which is determined by the probability of the algorithm successfully detecting a change point in the stream.
- **The Mean Time to Detection**, which is the speed at which an algorithm can indicate that a change has occurred after the underlying data stream changes.
- **Detected Weight** the sum of the weight of the changes that were successfully detected by the algorithm.

We apply weighting to the change points using a method described below. This is to grade the change points by how *important* they are considered to be, with respect to their impact on clustering results. It makes sense that not all distribution changes are equal, in that some may be trivial to handle using standard clustering algorithms. Thus, we are particularly interested in change points which have a greater effect on the performance of our chosen clustering algorithm over time. A change which prompts a large change in the precision and recall of the algorithm is considered far more interesting than one which has little to no effect.

Additionally in this paper we use a combined evaluation measure known as MTR - Mean Time Ratio (Bifet et al., 2013). MTR is a combination of the several important metrics listed above, 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:

$$MTR = \frac{MTFA}{MTD} \times (1 - MDR), \tag{3.14}$$

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 an evaluation metric because of its ability to clearly show a comparison between algorithms optimised for different things. Looking simply at the number of successful detections or the number of false alarms doesn't 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.

### 3.4.2 Dataset

**KDD Cup 99'** 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. The KDD dataset contains data points which represent normal traffic as well as others representing 22 different types of attacks with varying durations from a handful of data points to hundreds of thousands of data points. 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. This subsampled dataset contains approximately 10% of the data of the full dataset, excluding much of the normal traffic.

KDD has been used previously as an example of a real-world application 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.

There are a variety of changes that can occur in a data stream over time, with varying degrees of difficulty to detect. Classes in the stream can disappear or emerge over time. They can move by shifting alone one or more dimensions. They can split into multiple sub-classes, or two existing classes can merge together to form one. The density of classes may also vary, becoming more represented in future samples, or more sparse. The shape and size of existing clusters might also vary.

All these types of changes can easily be replicated in synthetic data streams. Unfortunately it is much harder to determine when changes occur in real-world datasets for the purpose of evaluation of our algorithm. Data streams that have ground truth data available to them are uncommon. For evaluation of our proposed method we have split the dataset into blocks of 1000 points. A stream distribution change point is defined

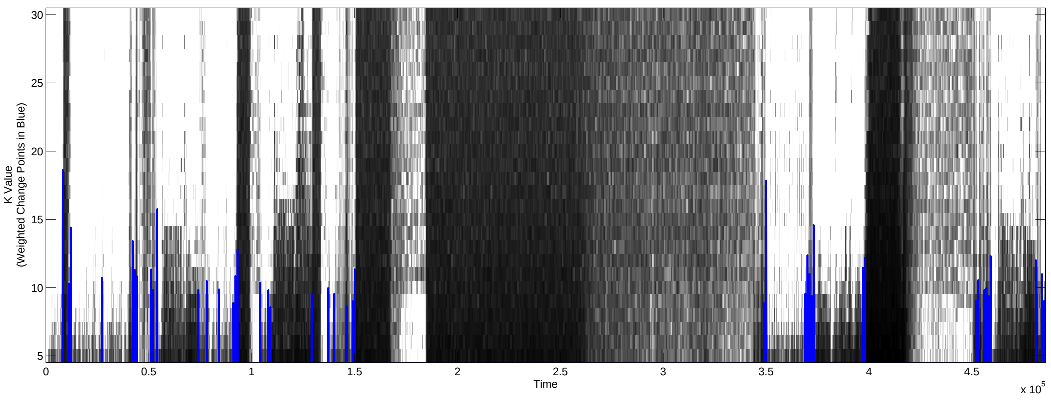


Figure 3.6: The heatmap of the KDD 99' Dataset, with change points marked.

whenever the set of classes present in one 1000 point block are not the same as the classes present in the next 1000 point block. This takes into account only certain types of change within the dataset, namely when classes appear, disappear, merge, or split. The dataset does not include information for when an existing class changes other than whether the class is present or not, but this is an expected limitation on evaluating real-world data, even with class labels to use in the evaluation process.

The heatmap for the KDD dataset is displayed in Figure 3.6. This Figure shows the heatmap in the background for the KDD dataset between $K$ values 5 and 30. In blue are marked where the change points occur, weighted according to the relevance measure defined above. The highest weighted change point has a weight of 14.14, and an average weight of 6.59.

**KDD Dataset Composition** The KDD data contains 23 different classes, including one class that represents normal data, and 22 which represent various types of network attacks or other suspicious network behaviour. The data was extracted by a smart firewall in a controlled environment and each data point was labelled depending on what the state of the network was at that particular time. The other classes include IP sweeps and port sweeps, scans using the Satan tool, unauthorised FTP transfers or unauthorised access from remote machines, or denial-of-service attacks. Despite there being a wide variety of attacks a huge majority of the dataset is comprised of just 3 classes - normal traffic, Neptune (SYN flood) DoS attacks, and Smurf (spoofed ICMP packets) DDoS attacks. Smurf attacks alone account for around 56.84% of the dataset, while the top 3 classes combined make up 98.23% of the data. This leaves the remaining 1.77% of the dataset being comprised of 20 classes.

| Attack Name | Count | Attack Name | Count |
|---|---|---|---|
| Normal | 97,277 | Perl | 3 |
| Back | 2,203 | PHF | 4 |
| Buffer Overflow | 30 | Pod | 264 |
| FTP Write | 8 | PortSweep | 1,040 |
| Guess Password | 54 | Rootkit | 10 |
| IMAP | 12 | Satan | 1,589 |
| IPSweep | 1,247 | Smurf | 280,790 |
| Land | 21 | Spy | 2 |
| Load Module | 9 | Teardrop | 979 |
| Multihop | 7 | Warezclient | 1,020 |
| Neptune | 107,201 | Waresmaser | 20 |
| Nmap | 231 | <**TOTAL**> | 494021 |

Table 3.1: Number of instances of each class in the KDD Cup 99' dataset.

Table 3.1 shows the number of data points labelled with each respective class in the entire data set. The Smurf DoS attack comprises more than half of the dataset alone, taking up a large section near the middle of the datase. Figure 3.7 shows the locations of the three most common classes - Smurf, Neptune, and normal traffic, other classes have been excluded from this diagram; included is also the heatmap for the dataset, showing where the clustering performance of RepStream varies. The Smurf attack (denoted by the grey sections at the top of Figure 3.7) is a kind of DDoS, which uses spoof ICMP packets to flood a target system with packets. In the KDD dataset this is characterised by a huge number of datapoints which have very little, if any, variation. During these attacks only two of the numerical features vary at all. This creates many instances of data points occurring in only a handful of locations in the feature space.

**Smurf Attack** Attacks like Smurf result in clustering algorithms reporting extremely high - even perfect - purity values, since in the regions where these attacks occur there is only a single class. No matter what the cluster configuration is for any window during these attacks the purity will always be 100%, as there are no other classes to cause impure clusters. However, judging by the heatmap in Figure 3.6 RepStream performs in general rather poorly during these regions. This is because of how RepStream handles multiple data points occurring at the same point. RepStream creates singularities which are clusters of these points which do not connect to any outside data points. This causes a low F-measure score because the singularities are not linked together into a single cluster, which lowers the resulting score. A notable occurrence, however, is at around $time = 1.8 \times 10^5$ where there is a region which suddenly has a much higher F-measure score. This is due to the network, at this time, stabilising and nearly every single data point in that region has exactly the same values for each feature. This results in a sudden increase in F-measure as all data points are correctly being placed into the same singularity cluster. Afterwards the variation resumes, which results in a boundary on the resulting F-measure value.

**Nepune Attack** The Neptune attack type (denoted by the white sections at the top of Figure 3.7) is a SYN flood DoS attack. This type of attack is much more varied in terms of feature values than the Smurf attack. Unlike that type of attack the Neptune class can become completely connected given a high enough $K$ value. This is the reason why RepStream has a much higher F-measure score on higher $K$ values during these regions ($K > 15$, near the top of the plot, results in extremely high scores). Traffic during these regions entirely consists of the single Neptune class, meaning the highest F-measure score can be achieved by linking all data points into one single large cluster. Due to the nature of the attack this requires a higher $K$ value, and thus during these regions the best performance is gained with $K$ values above 15. This is an ideal example of when a distributional change in the dataset requires that the $K$ value be at a specific level, or in a specific range. A static $K$ value is not optimal in terms of performance or outcome.

**Normal Traffic and Other Attacks** Normal traffic (denoted by the black sections at the top of Figure 3.7) is the only one of the three major classes where other classes are interspersed into the data points. As such the classes do need to be separated to reach optimal purity and F-measure scores. Normal traffic varies in many of its features, similar to how Neptune does, although in general the regions of normal traffic achieve better scores using lower $K$ values than what occurs during the Neptune attacks. This implies that normal traffic is easier to cluster, being able to be linked up even at very low $K$ values. A visualisation of this might look like an evenly-spaced cloud of data points, though such visualisation is not possible when 35 dimensions are present. Contained within the regions of normal traffic are all the other classes listed in Table 3.1. In total there are 97,277 data points of normal traffic, whilst all the other classes (excluding Neptune and Smurf) have a total of 8,753 data points between them, interspersed into the dataset.

Out of these remaining 20 classes 15 of them contain fewer than 1000 data points, and 12 of those contain fewer than 100 points. These types of very short lived classes

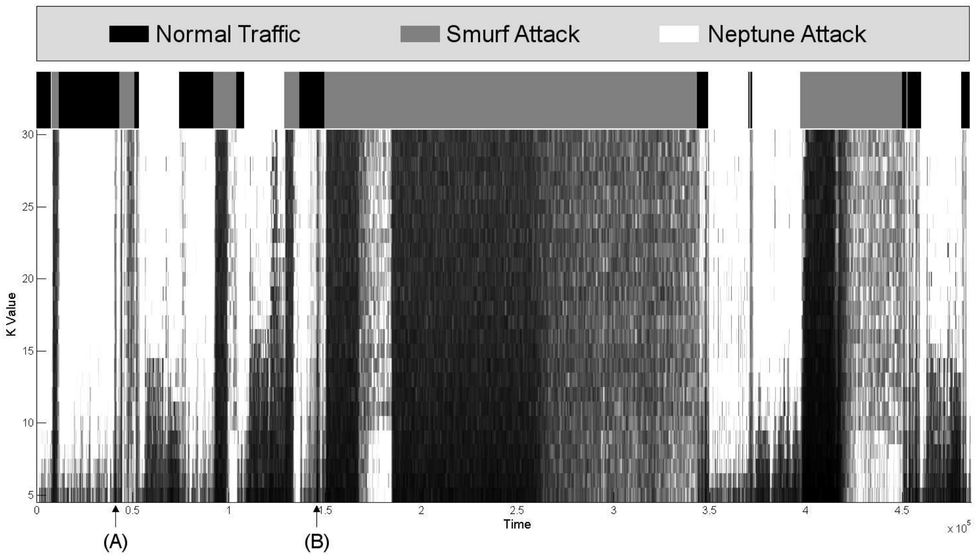


Figure 3.7: Locations of the three most common classes in the KDD Dataset, with heatmap.

are extremely difficult to detect if they are not significantly different from other existing clusters. Fluctuations caused by the sampling of data from the underlying distribution, or other sources of noise in the data may drown out these classes entirely, particularly in the cases where the data makes up less than 10% of the window size. Out of the remaining classes that have 1000 or more data points some occur immediately before or immediately after either Neptune or DoS attacks. The only exceptions are marked as (A) and (B) in Figure 3.7. At the location (A) 4 classes appear in close proximity - Back, IPSweep, Portsweep, and Satan. This causes a noticeable, albeit short, dip in RepStream's clustering accuracy, as various new classes appear in rapid succession, and it takes time to adapt. Another similar location occurs near (B) where a relatively high concentration of the Warezclient class is present, alongside the other classes that appear at (A). This also causes a noticeable dark streak on the heatmap.

All the transitions between the classes are what we use as the ground truth change points in our evaluations. The locations where major attacks occur represent distributional changes in the stream. The major change points, particularly between the three most prominent classes, are the most dramatic and important to detect since they affect the clustering and any analysis of the dataset. The goal, therefore is to find as many of these change points as possible using features extracted from RepStream.

### 3.4.3 Feature Evaluation

Figures 3.8 and 3.9 show the raw feature data as time series for the KDD dataset. This is the raw data which is fed into our change detection algorithm. When compared to the heatmap and change points shown in Figure 3.6 the regions of change and variation in the time series seems to match well with the ground truth changes. Both the heatmap and change points have been added to these figures for reference. There is a rather high level of variation in the time series in regions where change is expected, which leads us to believe these features are ideal for change detection.

The heatmap is shown in Figure 3.8 for illustrative purposes. Significant change points cause variation in the heatmap, as demonstrated earlier. History count shows significant and very obvious spikes during the first $1.5 \times 10^5$ points. There is especially a large amount of instability among the first $0.5 \times 10^5$ points, despite the fact that the ground truth changes occur at the very beginning and end of this region. However between $0.5 \times 10^5$ and $1.0 \times 10^5$ the ground truth change points are concentrated near the beginning and end, which is also reflected by the History Count time series. There are more changes towards the end of this region, and also more volatility to match.

The same thing again is shown between $1.0 \times 10^5$ and $1.5 \times 10^5$, however afterwards until $3.5 \times 10^5$ the time series is steady at a very low value, with little significant change compared to the rest of the time series. This corresponds to a region where no ground truth changes occur. Between $3.5 \times 10^5$ and $4.0 \times 10^5$ there is a lot of variation in the time series, with noticeable peaks and troughs at the three major points where change points occur. Towards the end of the time series there are two more clustered regions of ground truth changes, which prompt spikes in the time series. These make History Count seem to be a very sensitive feature, as it produces more prominent spikes at specific locations. This would lead to a higher detection rate, concentrated around regions of change.

History Count is defined as the number of combined times all points in memory have returned to a cluster that it was previously a part of. This number does not constantly increase, because of how RepStream discards older points over time. Still, the large window size leads to this feature being the highest in magnitude compared to the other features. This feature tends to decrease when the data distribution is static, as older points with larger history count values are removed from the window, and the feature increases when there are changes, since after a distribution change points tend to change cluster membership before stabilising. This is what we see in the History Count feature. The feature plot rises, sometimes very sharply, when a change occurs,

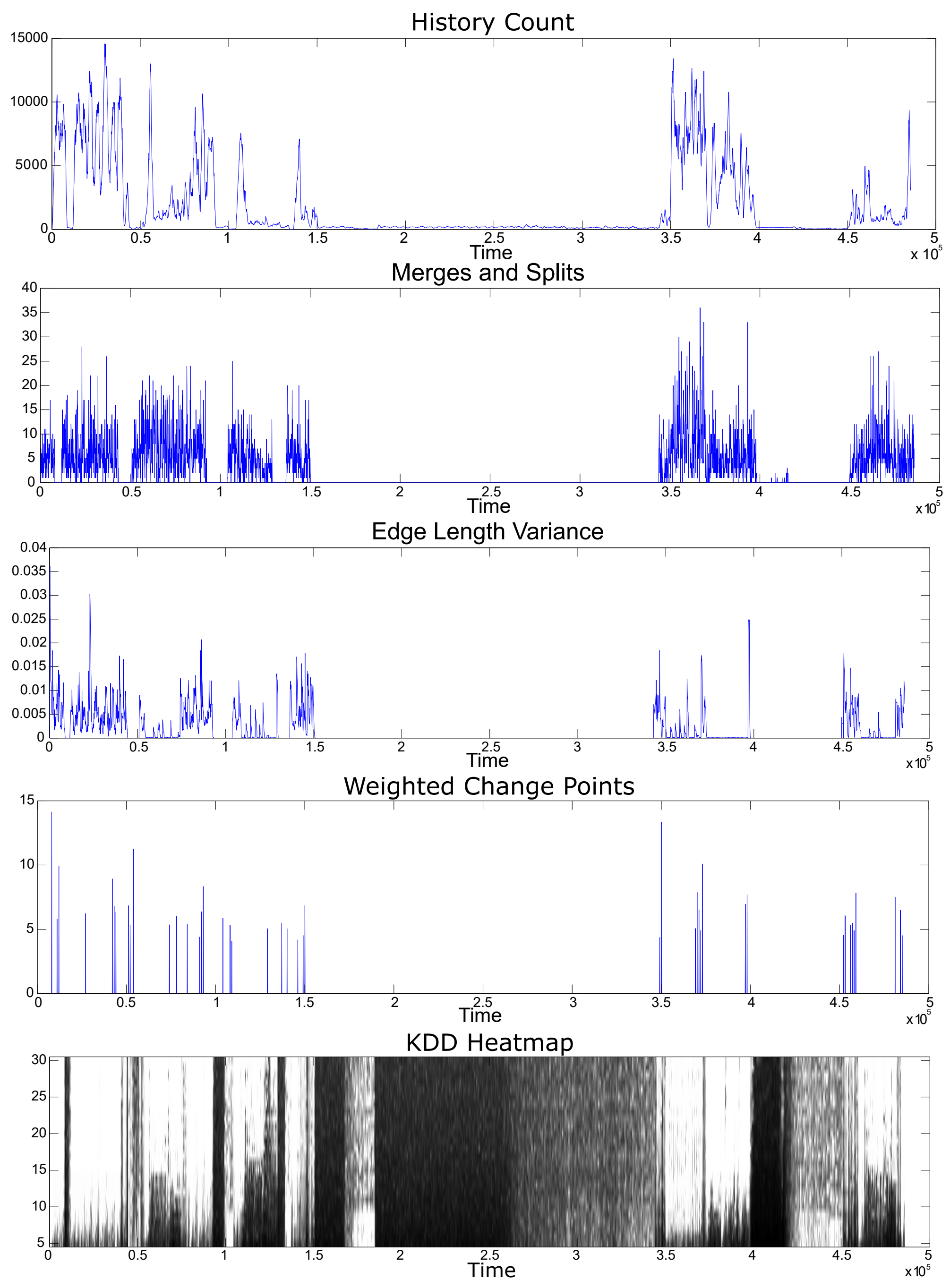


Figure 3.8: Raw feature data taken from the KDD dataset with the heatmap for comparison.

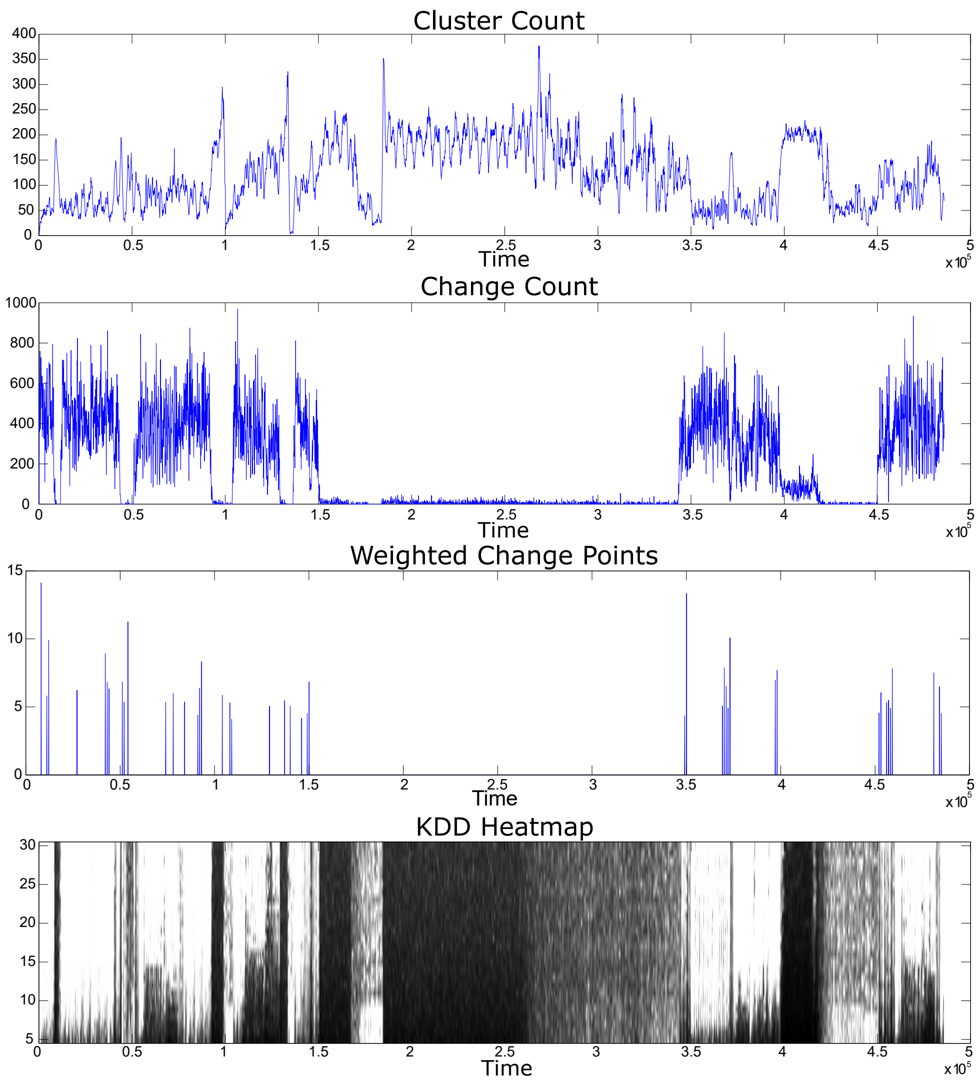


Figure 3.9: Raw feature data taken from the KDD dataset with the heatmap for comparison.

before returning to near zero over time.

The Merges and Splits plot in 3.8 is much more varied, producing a very noisy time series. Due to the high standard deviation in the time series it seems best at detecting slower changes. There are regions where the time series stabilises - noticeably around $0.5 \times 10^5$, $1.0 \times 10^5$, in the large region between $1.5 \times 10^5$ and $3.5 \times 10^5$, and also between $4.0 \times 10^5$ and $4.5 \times 10^5$. This implies regions where the dataset goes from being unstable to stable, and these would be detected as change points for this time series. In the more noisy regions there are some locations where the average seems to vary more, particularly between $3.5 \times 10^5$ and $4.0 \times 10^5$. The Merges and Splits time series, due to its high amount of noise, seems more suited to detecting major, or slow changes when the data is smoothed with a moving average. Merges and splits are extracted by counting the number of times a cluster joins another existing cluster, or when an existing cluster splits into multiple clusters. If the data is being pulled from a static distribution the expectation is that the number of clusters would remain relatively static. Conversely, after a change the algorithm will take time to settle to a stable clustering solution as points from the new distribution eventually replace all the older data. Because of this clusters will rapidly merge and split after significant changes, and remain fluctuating for some time until they eventually stabilise. This is the reason for the large amount of noise in the merges and splits feature.

The edge-length variance plot exhibits spiky behaviour similarly to History Count, with major spikes concentrated in similar regions. It produces more obvious peaks in the first $0.5 \times 10^5$ points, around the three major regions of ground truth changes. There are also some major spikes near other major change points, specifically around $0.8 \times 10^5$, $3.5 \times 10^5$, $4.0 \times 10^5$, and $4.5 \times 10^5$. Also as expected, in the middle, between $1.5 \times 10^5$ and $3.5 \times 10^5$ the time series is virtually flat compared to the rest. This time series exhibits the least noise out of all the features, and will result in a lower rate of detections. Because of the lower rate of detections it is more suited to reinforcing the other more sensitive features which produce a higher number of detections. Edge-length variance reflects the overall spread in the lengths of the edges in the whole dataset. When a new class emerges new points will be added, likely a significant distance from other data points, which should result in a spike in the variance plot. Of course, other things can also cause spikes, for example when points decay and are removed, or when a class changes. The various cases under which this particular feature can vary means there should be a lot of spikes, which is what we see in the plot itself. For this reason it would perform poorly for individual detection, but is useful for verifying other detections.

The cluster count plot time series, in Figure 3.9, appears almost cyclical, alternating steadily between peaks and valleys. This is, as expected, because of the decay of representative points over time in RepStream. However, the time series does show large shifts at locations where change occurs, at similar points to the other features. Notable are large shifts at $1.0 \times 10^5$, $1.4 \times 10^5$, $1.8 \times 10^5$, $2.7 \times 10^5$, and $4.0 \times 10^5$. All of these locations produce large rising amounts of clusters. This is likely to be caused by an unstable shifting dataset, as new smaller clusters are created where clusters previously didn't exist. These regions where there is a rapid increase in clusters presumably corresponds to the data distribution shifting in a dramatic way. Even when the distribution of the data points is stable the number of clusters in RepStream's memory varies. This is due to the noise created by the selection of data points from the distribution, which can cause even a stable distribution to fluctuate in the number of clusters in memory at any given time. The constant addition and removal of data points from memory causes the number of clusters to vary, however it will generally oscillate around a specific number if the distribution is stable. When changes occur one expects the number of clusters to change, generally increasing rapidly as data points are added in previously empty areas. After significant changes the number of clusters will begin to fluctuate around a stable point. This is reflected on the feature plot, where large changes cause spikes and sudden changes in the cluster count, even when it fluctuates steadily over time.

The edge change count feature is also a very noisy feature, much like the merges-and-splits time series. Edge change count does vary a lot from sample to sample, but over time tends to remain within a limited range so long as no changes are occurring. The distribution of the time series does shift rather rapidly at times, exhibiting changes in the distribution of the time series at similar points to the Merges and Splits plot. However, this plot seems to have more well-defined change points, specifically around $3.5 \times 10^5$ to $4.5 \times 10^5$ where several distinct ranges for the time series can be seen. Edge change count is defined as the number of times a representative point in RepStream's $K$-nearest neighbour graph structure has an outgoing edge on the representative layer added or removed. Representative points stay around longer than other points which are subject to the first-in-first-out queue used by RepStream. They represent nearby points in clustering decisions to make the algorithm more efficient. Because of this there is a lower amount of changes in their outgoing edges than standard points. New representatives are usually formed more frequently when the distribution changes, as new representatives will need to be added to represent areas that previously didn't have many data points. Because representative points eventually do decay there is always a certain level of change in the edges, even if it is only a small amount. In Figure 3.9 it is particularly obvious where the DoS attacks occur due to those regions having little to no change in the representative edges because of the data points being concentrated and virtually uniform.

All of the features in Figures 3.8 and 3.9 show change at similar locations, however some perform better at detecting change in different regions than others. Figure 3.10 shows detection done on each of the features individually. $H = 20$, $M = 1$, and $\lambda = 0.001$ were selected for demonstration purposes, as these were the values used during testing and debugging of the methods. We did not expect any single feature to perform best on its own as they are all designed for different purposes, however comparing the features individually to the locations of the ground truth changes - shown at the top in red - some features match up well enough to warrant further investigation.

Some of the features, particularly cluster count and history count are very volatile, and produce a large number of detections. This is unsuitable individually since that would lead to a large number of false alarms. On the other hand the other features (change count, edge-length variance, and cluster merges-and-splits) produce a lower number of detections, but not always in places that overlap with each other. This gives a chance that any of those individual features may miss out on change points that the other methods should detect.

Table 3.2 shows the evaluation of our detection being run only on single individual features. As such we have chosen a majority voting method between all of the feature detections for our final decision. Listed are the values for the number of false alarms (FA), number of correct detections (D) number of non-detected change points (ND), Mean Time to Detection (MTD), Mean Time Ratio as defined earlier (MTR), and the sum of the weight of the change points successfully detected by the algorithm (DW) with the total possible weight listed as (TW). The table shows, as we expect, History Count and Cluster Count are very sensitive, and produce a higher number of detections compared to the other features, whilst also having a much larger false alarm rate. The other features have lower false alarm rates, but also have lower detection rates. The detection done using just the Change Count feature seems to perform reasonably well all around, but of course isn't perfect.

As such we have chosen to use a majority voting method to combine the features

|  | FA | D | ND | MTD | MTR | DW | TW |
|---|---|---|---|---|---|---|---|
| History Count | 100 | **28** | **14** | **7.36** | 4.36 | **202.09** | 283.29 |
| Merges & Splits | **3** | 14 | 28 | 166.71 | 2.00 | 115.84 | 283.29 |
| Edge-length Variance | 5 | 10 | 32 | 18.20 | 6.85 | 87.61 | 283.29 |
| Cluster Count | 93 | 19 | 23 | 13.16 | 1.81 | 148.90 | 283.29 |
| Change Count | 18 | 21 | 21 | 11.10 | **11.33** | 162.83 | 283.29 |

Table 3.2: Results of change detection using only individual features.

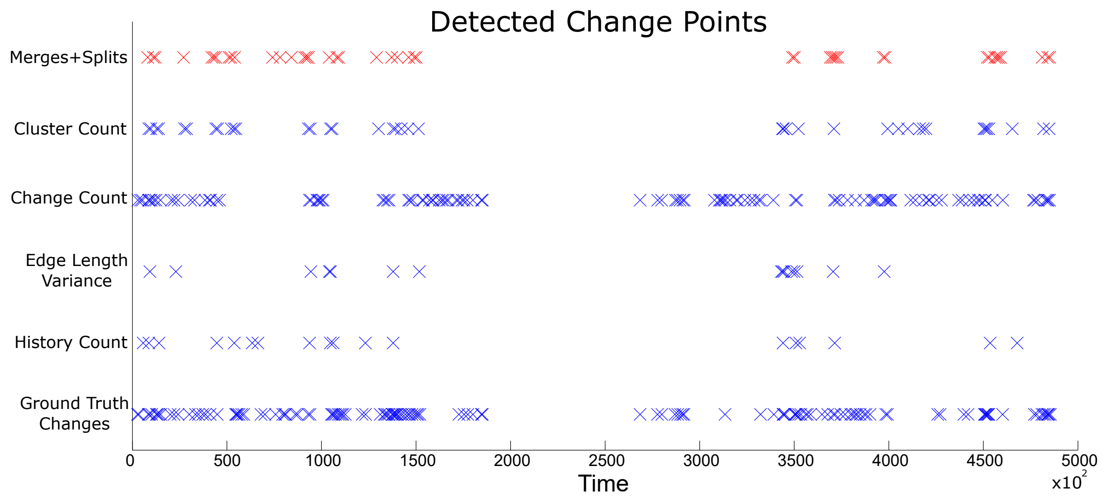


Figure 3.10: Detected change points using individual features.

together, and to attempt to capitalise on the benefits of each feature.

### 3.4.4 Setup and Parameter Selection

The features for all datasets were extracted from an instance of RepStream running at $K = 10$, $\alpha = 1.5$, a memory horizon of 1000 points, and using standard normalisation through the entire dataset. Though these parameters are not optimal through most of the dataset they seem to produce good change-detection results, which implies an optimal $K$ value is not required to do change detection, which is useful when the optimal $K$ value is not known. Features were extracted every 100 points and dumped into debug files.

We ran both our proposed method and our comparison algorithm multiple times, using specified ranges for the input parameters. Each algorithm was run using every possible permutation of values within the specified ranges, and the values which resulted in the highest combined Detection Rate vs False Alarm Rate was selected as the evaluation parameter values.

The parameter value ranges we used for our algorithm was as follows:
$M = \{0.8, 0.9, ...1.5\}$
$H = \{8, 9, 10, ...40\}$
$\lambda = \{0, 0.001, 0.002, 0.005, 0.1\}$.

We also ran the same experiments on the PCA Change Detection algorithm described by (Qahtan et al., 2015). They present a PCA-based change detection method for use in multidimensional vector data streams. This method computes the principle components of a window of data points and then defines the distribution of points in that projected dimension. A divergence metric is then used to compute a change score between a reference window and a test window. The raw dataset was input to the algorithm, which output a series of change points in a log file. These change points were evaluated using Matlab Parameters for the PCA-Change Detection method were tested between the ranges:
$w = \{100, 500, 1000, 2500, 5000, 7500, 10000, 15000, 20000\}$
$ThresholdFactor = \{50, 100, 250, 500, 750, 1000, 1500, 2000\}$
$\delta = \{0.001, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5\}$
We also use both available area metrics.

The parameters used for evaluation in the following section were selected by sorting all the results by MTR score in descending order. This ensures a good balance between detection rate, false alarm rate, and detection speed. From this list the set of results with the highest detection rates were selected, and from among those results the single entry with the most favourable outcomes were chosen as the final result.

For the KDD dataset the parameters selected for our proposed method are $M = 1.0$, $H = 14$, and $\lambda = 0.002$. This allows the algorithm to adapt steadily to slow changes, and maintains a window size which is 40% larger than the number of points that RepStream stores in memory. Our proposed algorithm depends on the parameters given to it. If the parameters are outside of certain ranges the algorithms will fail to produce any useful results. The parameters we have selected for testing performed reasonably well, however we have evaluated a range of parameters to optimise the performance.

For the PCA Change Detection method the parameters $WindowSize = 1000$, $ThresholdFactor = 100$, $\delta = 0.02$ were selected, with the 'Area Metric' divergence metric selected, which in the original paper outperforms the other metrics that were tested.

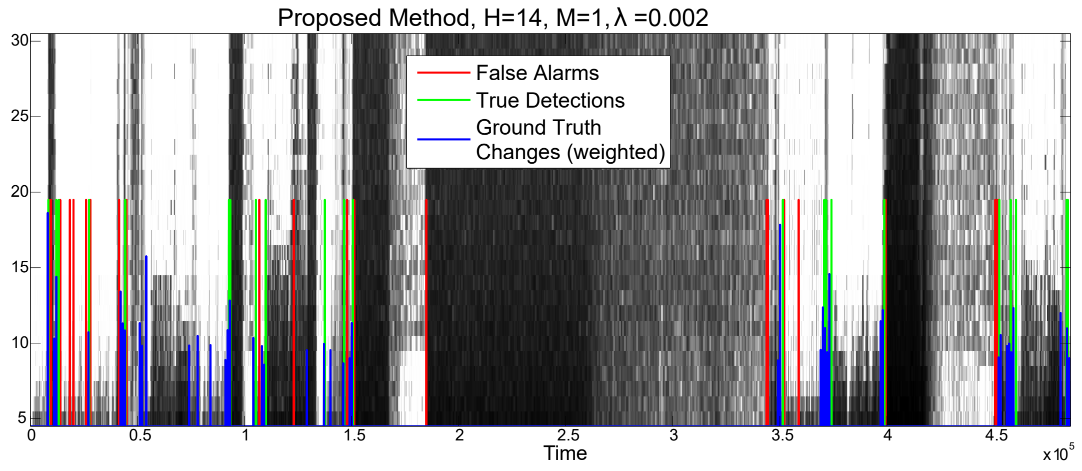


Figure 3.11: Our moving average detection algorithm on the KDD dataset.

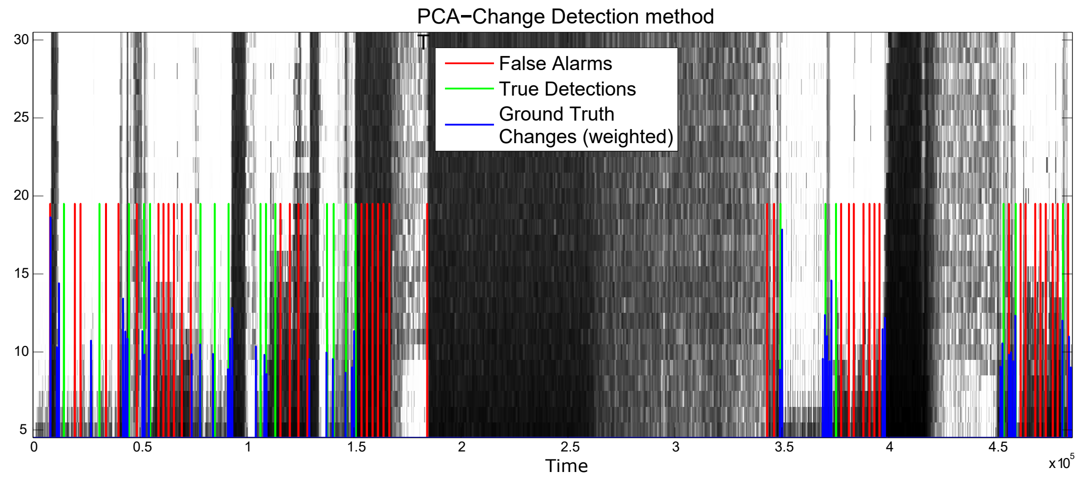


Figure 3.12: PCA Change Detection algorithm run on the KDD dataset.

### 3.4.5 Results

Figure 3.11, show the outcome of our algorithm on the KDD Network Intrusion dataset, using the Heatmap in the background for illustration. The blue lines represent the ground truth change points, weighted with our proposed method. Marked in green are where the algorithm has detected a true positive with the given parameters, marked in red are where our algorithm has detected a false positive. Figure 3.12 shows the results for the PCA-based Change Detection algorithm.

Figure 3.3 shows a table of the results for our detection algorithm. It contains values for the number of false alarms, number of correct detections number of non-detected change points, Mean Time to Detection, Mean Time Ratio, the detected weight for each method, and the total possible weight. On the KDD Dataset our proposed method performs better in all listed categories. It has a higher detection rate, a

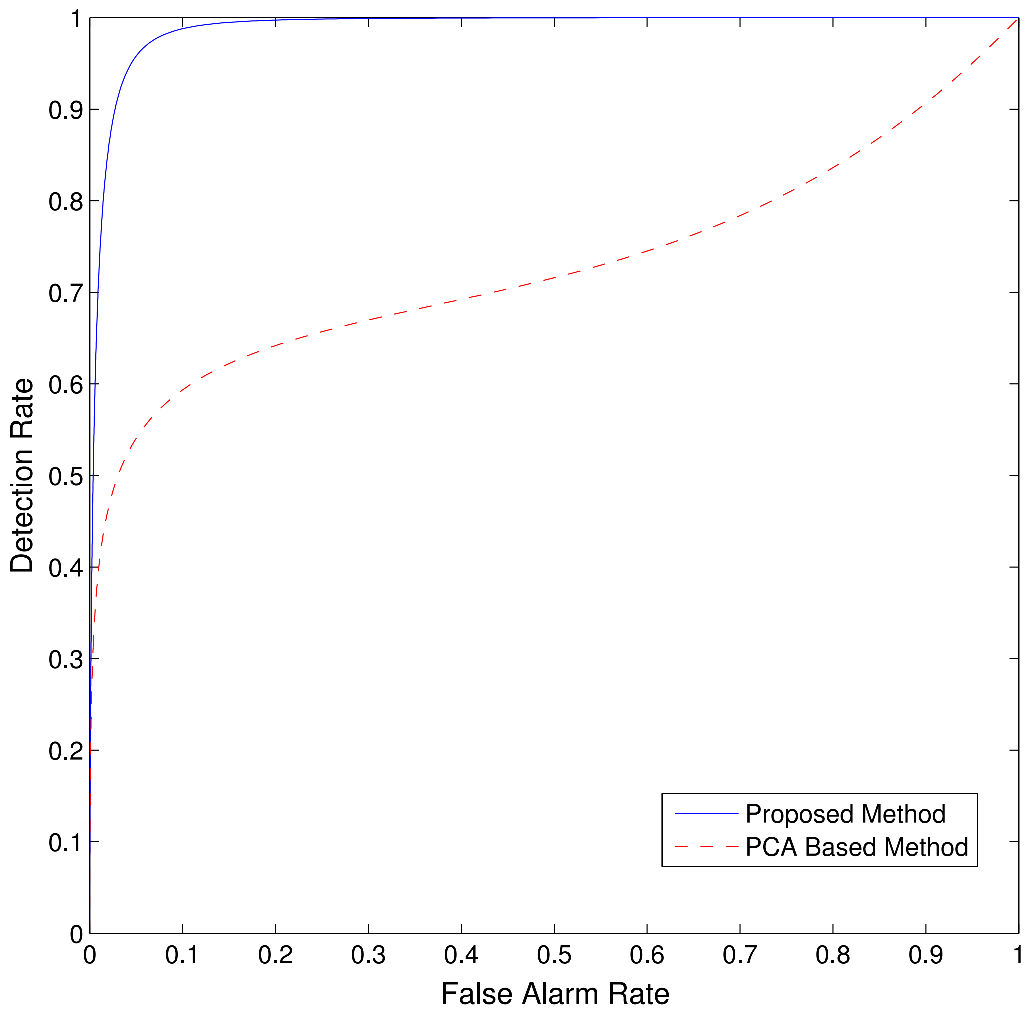


Figure 3.13: ROC Curve of the two methods, using curve-fitting. AUC is 0.9888 for the proposed method and 0.7298 for the PCA based method.

|  | FA | D | ND | MTD | MTR | DW | TW |
|---|---|---|---|---|---|---|---|
| Our Method | **22** | **22** | **20** | **6.55** | **16.45** | **165.47** | 283.29 |
| PCA-CD | 40 | 21 | 21 | 9.38 | 6.33 | 141.37 | 283.29 |

Table 3.3: Results of our experiments on KDD with the Mean Time between False Alarms, Mean Time to Detection, Missed Detection Rate, and Mean Time Ratio.

lower false-alarm rate, the mean time to detection is lower, and the total weight of the detected change points is higher.

Figure 3.13 shows the ROC curve for the two methods, displaying the true positive versus false positive rate. The area under the curve (AUC) for the methods is 0.9888 for the proposed method and 0.7298 for the PCA-based method. Generally speaking, a curve which has the largest AUC is preferable.

## 3.5 Discussion

The locations in the KDD dataset where our algorithm detects change match up well to the most important ground truth change points - that is, locations where there are higher weighted ground truth changes. Detections are missing only in areas where multiple change points occur rapidly. Our proposed algorithm has no way of dealing with changes that occur rapidly, which we consider to be a limitation, however other similar methods also have problems with detecting changes over short amounts of time.

Rapid changes are when multiple change points exist within a short time period of each other. On Figures 3.11 and 3.12 they would be represented by the places where ground truth changes occur in clusters, because of multiple classes appearing and disappearing rapidly. For our purposes we will consider any ground truth changes that occur less than one RepStream window length (1000 data points) from each other to be occurring in a rapid succession, or cluster. These groupings of change points are very difficult to detect as there is less time to adjust to the new distribution before the new distribution emerges. This creates a significant challenge to change detection algorithms, which may detect these changes as being a single change instead.

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, yet is a realistic representation which demonstrates the challenge of handling evolving stream data, even when labels are present.

### 3.5.1 Qualitative Analysis

In terms of accuracy our method produces relatively fewer false alarms in total than our comparison method.

During the first $0.5 \times 10^5$ data points we produce 9 false alarms. This section is a higher false alarm rate than the PCA change detection method, however it is also the highest concentration of false alarms that there is through the whole dataset. There are 5 correct detections within this region, detecting both the start and the end of the short Smurf attack at $time = 0.1 \times 10^5$. Between $Time = 0.5 \times 10^5$ to $1.5 \times 10^5$ the false alarm rate drops much lower, producing only 3 false alarms during this time, while producing 6 correct detections. These correct detections occur at the beginning and ends of the Smurf and Neptune classes, shown in Figure 3.7. These are the two major classes which are most prominent in the dataset and are most distinct, and thus easy to detect.

Two of the false alarms within this region occur near other true detections, just before $Time = 1.1 \times 10^5$ and $1.5 \times 10^5$, whilst the remaining false alarm occurs in the middle of a Neptune attack. Neptune is a class which is more varied than Smurf attacks, and is harder to analyse because of its higher variance. In this situation our algorithm detected fluctuations in the class as a change point. This occurs later again at $Time = 3.6 \times 10^5$, which shows how the Neptune class causes some problems. Regardless, our algorithm detects the start and end points of the Neptune class instances fairly well, missing only the region that start just after $0.5 \times 10^5$ and the region ending at $1.3 \times 10^5$.

At $Time = 1.5 \times 10^5$ we correctly detect the the beginning of the large Smurf attack in the middle of the dataset, however the method produces another detection soon after, which is a false alarm, as it adjusts to the new distribution. Change points produced in rapid succession are likely to be a result of over-sensitivity, and are a drawback.

There is also a false alarm produced at $Time = 1.8 \times 10^5$ where the distribution of the points in the Smurf attack does shift slightly, as noted earlier. While we count this as a false alarm, as the actual class does not change, it is worthwhile to note that this is a type of distribution change, as classes can vary in their exact composition over time.

The rest of the region ending at $Time = 3.4 \times 10^5$ correctly produces no detections, as the class in this region has little variance. Following this, between $Time = 3.5 \times 10^5$ and $4.0 \times 10^5$ there are two periods of the Neptune attack, with a short but noticeable break in between. The transitions into and out of these regions are all detected correctly, and the sensitivity of our algorithm even detects multiple of the smaller change points at $3.7 \times 10^5$. From $4.0 \times 10^5$ to $4.5 \times 10^5$ there is another significant Smurf attack, which we correctly detect the beginning and end of, though with a false alarm near the end, as it detects the change in the wrong place.

Other than the main 3 classes which make up more than 90% of the dataset there are 20 other in the KDD dataset which appear occasionally. As mentioned before only 8 of these classes contain more than 100 data points, and only 5 classes contain more than 1000. We refer to all of these classes as minor classes. These classes tend to occur in clusters, where multiple of them appear within a short amount of time of each other. There are concentrations of classes like this at $0.395 \times 10^5$ to $0.430 \times 10^5$, where 9 of these classes appear within around 3500 data points. There is another at $0.509 \times 10^5$ where 5 classes occur within 2000 points.

At $0.741 \times 10^5$ there is a large collection of 14 clusters, however this region spreads over more than 30,000 data points, so the data points for these classes are too spread out to be easily detected. However at $1.400 \times 10^5$ there is another large concentration of classes, including one of the most populous minor classes. In this region there are 12 classes active over a 10,000 point region, however one of those classes contains just over 1000 data points in this region. This region creates a noticeable dark region on the heatmap, which indicates RepStream has trouble adapting and classifying in this region. Our change detection method, however, does successfully detect a change point at this location.

There is also a cluster of minor classes at $3.440 \times 10^5$ and $3.703 \times 10^5$, both which coincide roughly with the ending points of major DoS classes. Scattered through the rest of the dataset are some more data points belonging to these minor classes, however they are spread out with a larger amount of normal traffic in between them. These ones are not detected by our method, except when they coincide with the changes in the major classes.

As expected, the minor classes are less likely to be detected. There are far fewer samples, some classes not even comprising enough to fill 10% of our selected window length even when not spread through the dataset. Some of these data points also may be mistaken for normal traffic as they have a smaller amount of difference compared to the large and relatively distinct major DoS classes.

### 3.5.2 Comparison Method Analysis

The PCA-based change detection method we compare against has many more false alarms distributed over time than our method. Looking at Figure 3.12 we notice an almost even spread of false alarms from $Time = 1.85 \times 10^5$. Between $Time = 0.0 \times 10^5$ to $0.5 \times 10^5$ there are 6 false alarms which are spread fairly evenly across the time period. The 3 detections which are present seem to match up poorly to the change points, meaning that they are late detections.

During $Time = 0.5 \times 10^5$ to $1.0 \times 10^5$ the false alarms are concentrated in the first half of this section. This corresponds to the Neptune attack region, which seems to be a common theme. During all the Neptune attack regions, shown in Figure 3.7, the PCA change detection generated detections (mostly as false alarms) at a high rate, with the exception of the region between $Time = 3.5 \times 10^5$ to $3.7 \times 10^5$ where it correctly detects nothing. During all other Neptune attack regions a high number of false alarms are generated. This is just what happens between $Time = 1.0 \times 10^5$ to $1.5 \times 10^5$, as the 4 false alarms generated during this time period are also when the Neptune class is present.

This implies that Neptune is a more difficult class to detect, because it has a larger amount of internal change by itself. This is something we expect based on our analysis of the class previously in Section 3.4.2.

During the period $Time = 0.5 \times 10^5$ to $1.5 \times 10^5$, however, the PCA change detection method does correctly locate 11 of the change points within this region, with a change point being in every region where change occurs (when rapid changes are grouped together), except for one change point at $Time = 1.7 \times 10^5$.

During the large Smurf attack in the middle of the dataset the PCA change detection method makes 6 detections over a short period from $Time = 1.5 \times 10^5$. These are false alarms caused by the shift in distribution as the new class takes over the dataset. This detection method has trouble adjusting to the change in this situation and thus report changes until the distribution changes again at $1.7 \times 10^5$, but remains in the same class. The algorithm then does not detect any changes for the remainder of the Smurf attack until $Time = 3.4 \times 10^5$.

Following this there are 2 sequential instances of Neptune attacks, followed by another instance of the Smurf class at $Time = 4.0 \times 10^5$. During the Neptune attacks, as noted before, a high number of false alarms are detected in the latter half, and again at the Neptune attack near the end of the dataset. However, despite the high amount of false alarms within this region, the PCA change detection method does detect all major change points, except the transition into the Smurf attacks at $4.0 \times 10^5$.

### 3.5.3 Parameters

There are three main parameters used in our proposed method – these are $M$, $H$, and $\lambda$.

The parameter $H$ is the size of the sliding window, which we refer to as the 'horizon' of samples in the feature. It is the number of points which we calculate our moving average over. Increasing this parameter results in the algorithm needing changes to be over a greater number of samples to detect them. A lower $H$ value means the algorithm is more sensitive to fluctuations and noise, but detects more short-lived changes. By varying this parameter it is possible to control the granularity of detection – knowing more about the kind of changes in the dataset helps setting this parameter.

The $M$ parameter is the multiplier used when determining whether a change has occurred. If the moving average is more than $M$ times the standard deviation away from the current region we detect a change point. This $M$ value affects the sensitivity of the algorithm. With a high $M$ value a change needs to be of a greater magnitude to be detected, while if $M$ is smaller the algorithm will detect less extreme changes. The user must be careful not to set this value too small, or noise in the datasets will result in changes being detected. On the other hand, if the $M$ value is too small the changes in the features may never be great enough to be detected. A value somewhere around 1.0 seems to produce good results for the KDD dataset, though it can be varied for different purposes.

Finally, $\lambda$ determines how fast the algorithm will adapt to slow changes. If $\lambda$ is set to 0 the algorithm will not adapt to slow changes at all, in this case even the slowest change in the feature time series will eventually be detected, given enough time. If $\lambda$ has a positive value the algorithm will adapt to slow changes, so slow changes are less likely to be detected. In this case the algorithm is more likely to detect only large and sudden changes, which ignores slower trends in the dataset.

### 3.5.4 Summary

Whilst our proposed algorithm performs well by comparison, it still has limitations, notably in the form of false alarms and changes which are not detected. For example the false alarms at around $Time = 1.85 \times 10^5$ and $Time = 1.92 \times 10^5$. 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 3.6). 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.

Our approach produces good detection of detecting the start and end points of the major class changes, as described above. These are the most extreme changes which have the largest impact on the dataset. Due to the nature of these changes they are also the easiest to detect because the distribution change is so great compared to the other classes. This is because the changes exist in large groups, the longest being more than a hundred thousand data points in length, and because the structure of the data changes so dramatically - in the case of the Smurf class to being many data points in only a handful of points in the feature space. It is also successful at keeping a relatively low false alarm rate, despite the noise evident in the dataset.

Despite the false alarms our algorithm detects a high amount of the ground truth changes (Table 3.3). 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 changes happen very rapidly. If a change is detected, and the change continues then the algorithm will report multiple changes occurring in a short time period. This can be considered a false alarm, as only one prolonged change is happening, rather than multiple changes. Additionally, if multiple change points occur within a short time period, for example less than the algorithm's parameter $H$ number of samples then the algorithm may not be able to detect the changes quickly enough, because it uses a moving average, which may be skewed by older data.

Interestingly, in the KDD data some of the false positives occur at locations where there is no ground truth change points yet where the heatmap reflects a change in the clustering performance of RepStream at that time - most notably the false alarms at approximately $T = 1.1 \times 10^5$, $T = 1.4 \times 10^5$ and at $T = 1.8 \times 10^5$. It is due to changes in the distribution of the dataset, whilst the ground truth class labels remain the same. This is something that is positive, since we are not merely seeking to detect the appearance and disappearance of classes, but changes in distribution that might affect clustering performance.

Another limitation of our algorithm is that the change points which are a result of only a small number of data points are almost impossible to detect. The major change points occur when one of the largest three classes in the dataset appears or disappears, however the other, less common change points making up less than 2% of the total data occasionally still have a noticeable effect on clustering performance for the time they are present, yet our algorithm fails to reliably detect them. This is not unexpected as larger more dramatic changes are, in general, much more easy to detect.

Our method is able to detect most of the major changes, with relatively few false alarms, given the highly complex data set. The actual data itself can fluctuate rapidly at times, yet with tuning our method can find the most important changes. Its higher number of true detections and the higher total detected weight reveals that it performs well in its task.

## 3.6 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. The features that we extract represent the geometric structure of the data in a $K$-nearest neighbour context, and we have found experimentally that change in these features corresponds to change in the distribution of a dataset.

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 method uses a non-traditional approach by looking at an abstracted version of the dataset which has been processed by a clustering algorithm. This differs from other methods which define a distribution in a reference window and look for changes in a later window.

Our detection algorithm outperforms a similar approach using PCA with respect to both its detection rate, false alarm rate, and time to detection 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 real world data. The approach is able to detect changes which have a greater effect on the clustering algorithm.

Our proposed method's ability to effectively locate significant change points within the data stream is an important step to our research. Knowing when the structure of a data stream changes allows us to locate potential locations when changing the operating parameters of a clustering algorithm may be beneficial. In our next chapter we will concentrate on using this knowledge to determine what the parameters should be changed to when the data stream changes. This will lead to higher quality clustering results from the existing algorithm - RepStream, and could potentially be generalised to include other stream clustering algorithms.

# Chapter 4

# Parameter Selection Using Edge Distribution Score In RepStream

## 4.1 Dynamic *K* Selection Overview

A common feature amongst even newer clustering algorithms is that they require user-set parameters to perform their clustering (Bhatnagar et al., 2014; Forestiero et al., 2013; Zhou et al., 2008). These user-set parameters affect how the clustering algorithm in question handles the data as it arrives from the stream, for example how the formation of core-micro-clusters are affected by the $\varepsilon$ radius parameter and $\mu$ density threshold in DenStream (Cao et al., 2006). Another example is D-Stream (Chen & Tu, 2007) which is a grid based clustering algorithm, using a grid granularity parameter $len$ and threshold parameters $C_m$ and $C_l$ to determine when cells in the grid are sparse, transitional, or dense. Parameters like these can greatly affect the output of the algorithms, and if set poorly can result in low quality output. Table 4.1 shows a selection of some well known and contemporary algorithms, as well as the input parameters which must be specified at runtime.

In the previous chapter we demonstrated how changes in distribution, which have a great effect on clustering output, can be detected by analysing features computed from internal representations of the data. In this chapter we seek to build on this idea and present a way to adjust the internal state of our clustering algorithm in order to produce higher quality clustering output with less reliance on user-selected parameters.

We discussed in Chapter 1 that the problem of parametrising algorithms is even more of a challenge in a stream clustering context. Whereas batch data is expected to have a single distribution which a clustering algorithm is attempting to find, a data

Table 4.1: Table showing a selection of parameters used by some contemporary clustering algorithms.

| **Algorithm** | **Params** |
|---|---|
| DenStream<br>(Cao et al., 2006) | $\mu$,$\beta$ Density Params<br>$\varepsilon$ Distance Param |
| D-Stream<br>(Chen & Tu, 2007) | $C_m, C_l, \beta$ Density Params<br>$len$ Grid Granularity<br>$\lambda$ Decay Param |
| BEStream<br>(Wattanakitrungroj et al., 2018) | $\Delta, \tau$ Density Params<br>$\lambda$ Decay Param<br>$\xi$ Distance Param<br>$\theta$ Direction Param |
| ADStream<br>(Ding et al., 2016) | $\xi, \varepsilon$ Density Params<br>$\lambda$ Decay Param |

stream can have the data distribution change over time. Naturally, this means that selecting initial parameters is challenging, but additionally an algorithm that has data-dependent input parameters can face the problem of the selected values being inappropriate later during a stream. Even if a user could guarantee optimally selected input parameter values initially, issues such as concept drift (Widmer & Kubat, 1996) in a stream mean that they could result in poor quality clustering at later points in the stream as various changes and evolution occurs.

In this chapter we propose an extension to the RepStream algorithm (Lühr & Lazarescu, 2009) which will allow the primary input parameter, the $K$ value, to be automatically varied over time in response to the structure and distribution of the incoming data. The RepStream algorithm works by representing data in a $K$-nearest neighbour directed sparse graph form, and creating outgoing edges from each data point to its $K$ closest neighbouring data points, according to a selected distance metric. The $K$ parameter, therefore, has a big impact on how the data is clustered together, as a higher $K$ value means a more connected graph, while a lower $K$ value results in a graph with fewer edges in it. Setting the $K$ parameter is vital to having high quality clustering output in the RepStream algorithm. Our proposed method allows RepStream to self-adjust to changes in the distribution of the data stream, allowing for recovery if the parameter is set poorly initially, as well as being able to adjust the value to a more appropriate value over time in response to changes in the stream's distribution.

A data stream is defined as a set of $d$-dimensional data points $X_1, X_2, ..., X_m...$ arriving at time stamps $t_1, t_2, ..., t_m...$ where a data point $X_i = [x_i^1, ...x_i^d]$ is a $d$-dimensional vector (Kaur et al., 2015). The data stream is potentially unlimited in length and data points are sampled in an unpredictable way from a data distribution that can change and shift over time. Because of the nature of data streams as being evolving and unpredictable the input parameters for clustering algorithms, and in our case RepStream, may be of varying usefulness at different times during the stream. Parameter values which initially may provide high quality clustering output may at a later point in the stream be less optimal than different values because of how the data evolves. RepStream in specific requires its $K$ parameter to be set such that the level of connectivity between data points is not too high - resulting in regions of data being grouped together when they should not be, and also not too low - which results in the algorithm fracturing the data points into too many small clusters. Even within the same dataset the optimal $K$ value can change over time. If an inappropriate $K$ value is selected then RepStream cannot recover.

Selecting an appropriate $K$ value is difficult if one has no prior knowledge of the dataset, and even such knowledge, if available, might be of no help when setting an input parameter for a clustering algorithms due to evolution over time in the data stream. Data clustering is an exploratory and unsupervised process (Jain et al., 1999), relying on internal validation metrics, and so no knowledge can be assumed before applying the algorithm to a given data stream. We consider high quality clustering output to be when the clustering algorithm is able to accurately cluster contiguous groups of roughly uniform density data points which are separated from each other by a region of different density data points, or regions of space containing no data points.

Our proposed method involves using a computed measure that we call the *edge distribution score*, which reveals some information about the properties and distribution of the data points. The edge distribution score is a measure which is intended to estimate the theoretical distribution of the edges connecting to nearest neighbours, and whether increasing or decreasing the $K$ value might be appropriate. By computing and measuring the average edge distribution score computed across all data points in memory we present a method for tuning and adjusting the $K$ value in RepStream dynamically over time. Starting from a given initial $K$ value we show that our method successfully allows the algorithm to adapt to changes in the stream, yielding higher quality clustering results, and without the need for the user to tune the $K$ parameter to the data stream itself. We show that our proposed method allows the algorithm to recover and produce high quality results even if an initially poor value for $K$ is specified.

This chapter's contributions are as follows:

- A measure known as 'edge distribution score' which we extract from the $K$-nearest neighbour sparse graph structure of RepStream.
- A method of $K$ selection in RepStream based on analysis of the edge distribution score across multiple $K$ values.
- Evaluation of the $K$ selection method on synthetic and real-world datasets to evaluate its performance in comparison to base RepStream as well as other stream clustering methods.

The chapter is organised as follows. Section 4.2 explains the concept of the edge distribution score, and details our $K$ selection method. In Section 4.3 we test the selection method on synthetic datasets and the well known benchmark KDD and Tree Cover Type datasets. Section 4.4 is a discussion and analysis of the results, and Section 4.5 is a summary and conclusion of the chapter.

## 4.2 Proposed Method

To select suitable $K$ values over time we extract and analyse a feature in the $K$-nearest neighbour structure of RepStream which we call *'edge distribution score'*. This computed feature is then fed into an incremental algorithm to determine which $K$ value to use at each time step. The edge distribution score of a graph is a measure which will reflect the rough distribution of nearest neighbour data, and gives us an idea of whether there exist outgoing edges from a vertex which connect to data points belonging to separate ground-truth classes. We wish to minimise the number of such edges, which we refer to as inter-class edges, while maximising the connectivity of data points belonging to the same theoretical ground-truth class - so called intra-class edges.

### 4.2.1 Inter versus Intra Class Edges

For our purposes in this thesis we use the term *classes* to refer to the theoretically perfect groupings of data points as determined by the distributions in the stream. Information on these ground-truth classes is typically not available because cluster analysis deals with unlabelled data, however evaluation and test datasets can be produced which have information on the ground-truth classes. In this chapter we refer to classes as the theoretically perfect cluster groupings, which are unknown to the algorithm and to the user. We also refer to clusters, which are the groupings of data points produced by the algorithm on demand during its runtime. These clusters are not necessarily the same as the ground truth classes, but if the information is available then an external validation measure can be used to determine the accuracy of the clustering, which we do in Section 4.3.

In cluster analysis we wish to find regions of data points in a more or less uniform density separated by space at a significantly different density of data points. Most often this takes the form of clusters of data points in specific arbitrarily shaped regions of the data space, separated by regions of empty space. This empty space between classes , which may sometimes contain noise data points,

We will define inter-class edges and intra-class edges as follows:

- **Inter-Class Edges** are edges which connect between two vertexes belonging to different classes.

- **Intra-Class Edges** are edges which connect between two vertexes belonging to the same class.

Figure 4.1 shows examples of these in a $K$-nearest neighbour graph where there are two ground-truth classes, and each vertex is a member of one of the classes. $E_1$ is an edge which connects two points belonging to different classes, and thus it is an inter-class edge. $E_2$ is an edge which connects two points belonging to the same class, and is an example of an intra-class edge.

As the $K$ value of a $K$-nearest neighbour graph increases, the vertices become more and more connected together. Density and graph-based clustering methods assume that there is a level of separation between classes which is greater than the distance between nearest neighbouring vertexes in the same class. Without this level of separation the boundaries between clusters would be virtually impossible to determine. This level of separation means that at lower $K$ values intra-class edges are more likely to form than inter-class edges. As the $K$ value increases the edges will connect to more and more distant vertexes, and inter-cluster edges become more common. The ideal $K$ value is one which connects the vertices of each class as much as possible, while avoiding connections between classes, so that they are not merged into the same cluster. In other words, we want a $K$ value which has many intra-class edges, and few inter-class edges. Some degree of inter-class connection is acceptable and will not result in merging clusters belonging to different classes, however the more inter-class connections

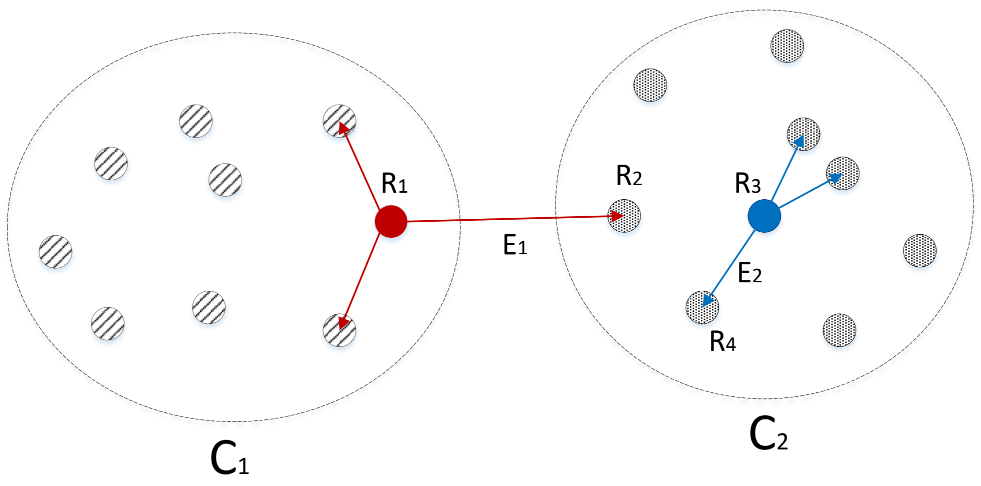


Figure 4.1: Intra and Inter-class edges. The Edge $E_1$ is considered an inter-class edge as it connects two vertices $R_1$ and $R_2$ that belong to different ground-truth classes. Edge $E_2$ connects two vertices belonging to the same class and thus is considered an intra-class edge.

can be avoided, the less likely such an event becomes.

Obviously these definitions of intra-class and inter-class edges require knowledge of the ground truth classes. This is something that is not known when performing clustering, so measuring intra-class and inter-class edges must be done indirectly. We propose a method of approximating the presence of inter-class edges by measuring a feature known as *edge distribution score*.

### 4.2.2 Edge Distribution Score

The edge distribution score is a feature we extract from the $K$-nearest neighbour graph structure. It is designed to give us an idea of the distribution of the edge lengths of a given vertex compared to how we would expect it to be in a normal clustering context. Edge distribution score gives us a measure calculated on each vertex which is determined by the relative edge lengths represented as a one-dimensional distribution. When this distribution is not consistent with our expectations of a stable cluster, the measure gives an indication of whether the number of outgoing edges is too high or too low.

Each vertex in RepStream has a number of outgoing edges which link to other vertices in a $K$-nearest neighbour fashion. To compute our distribution score we use the edge lengths of these outgoing edges, and treat them as a one-dimensional distribution.

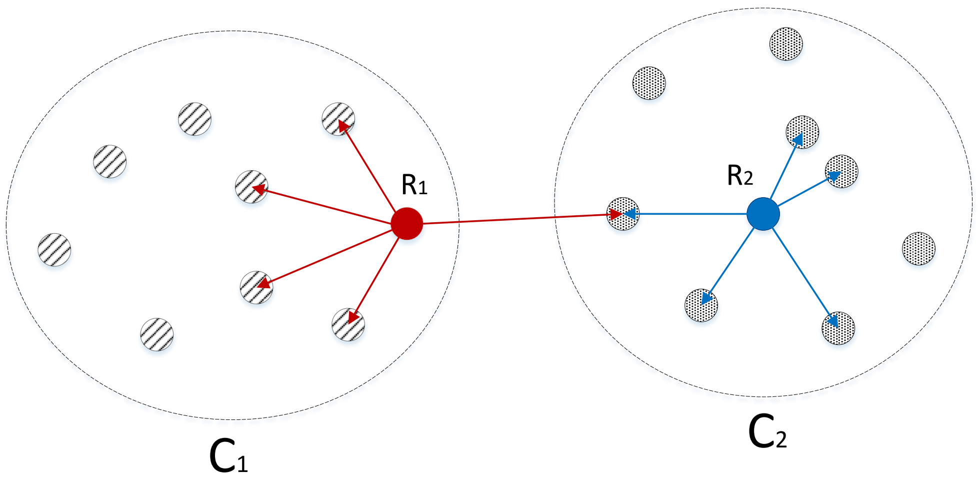


Figure 4.2: An illustration of relative edge lengths of nearest neighbours in the middle of a cluster versus near the edge of a cluster.

In these computations the direction and relative position of the edges is not taken into account, only the length of each edge.

In what follows, we denote $v_i$ as the vertex of node $i$ in the $K$-NN graph, and $e_i^1, e_i^2, \dots, e_i^K$ as the outgoing edges from node $i$. The length of an edge $e_i^j$ is $l_i^j$ We introduce the following concepts

**Definition 4.2.1** *The span $s_i$ of vertex $v_i$ is the edge length to its farthest node within its K neighbourhood $\mathcal{N}_K(v_i)$ is*

$$s_i = \max_{j \in \mathcal{N}_K(v_i)} l_i^j. \tag{4.1}$$

The span $s_i$ of a vertex is a non-negative quantity and it represents the support of the local neighbourhood. Denote $r_i$ as the interquartile range, $m_i$ as the median, and $\mu_i = \frac{1}{2}(l_{\max} - l_{\min})$ as the average of the shortest and longest of the edge length values associated with a vertex $v_i$. We formally define the edge distribution score (EDS) as follows

**Definition 4.2.2** *The edge distribution score of a vertex $v_i$ is given by*

$$EDS(v_i) = \begin{cases} 2\theta & \textit{if } s_i = 0, r_i = 0 \\ -1 & \textit{if } s_i > 0, r_i = 0 \\ \frac{2(m_i - \mu_i)}{r_i} & \textit{if } r_i > 0 \end{cases} \tag{4.2}$$

*where $\theta$ is a fixed threshold independent of the data stream statistics.*

In the above definition, the first branch accounts for the rare case when all vertices within the $K$-neighbourhood of a vertex are the same as the vertex itself. In this case, the distribution of the edge length reduces to a singularity. The second branch accounts for a similar and rare case where a large proportion of the vertices within the $K$-neighbourhood are the same and cause the inter-quartile range $r_i$, which is a measure of the spread of the distribution, to become zero. The third branch is what we expect most common: the distribution is either left skew, right skew, or symmetric.

From the definition, it can be seen that EDS is a measure similar to the skewness of a distribution. The difference is that our definition caters for the special case, and the mean value in the usual skewness has been replaced by the average of the extreme values of the edge length.

Let $\mathscr{G}_K(\mathscr{V},\mathscr{E})$ be the $K$-NN graph consisting of the vertices $\mathscr{V}$ and edges $\mathscr{E}$. We introduce the following definition

**Definition 4.2.3** *The average distribution score (ADS) for an K-NN graph $\mathscr{G}_K(\mathscr{V},\mathscr{E})$ is defined as*

$$ADS(\mathscr{G}) = \frac{1}{|\mathscr{V}|}\sum_{v_i\in\mathscr{V}} EDS(v_i) \tag{4.3}$$

*where $|\mathscr{V}|$ denotes the total number of vertices in the graph.*

We refer to distributions where the median edge length is greater than the midpoint value as being *right-heavy*, while when the median edge length is less than the midpoint value we refer to it as *left-heavy*. For a vertex surrounded by roughly evenly distributed vertices we expect that the distribution of edge lengths for the outgoing edges will be right-heavy. This is due to the fact that the volume increases exponentially when radius increases, when the data is 2 dimensional or higher. This leads to a situation where given an arbitrary radius $r$ from a vertex $v_i$, which forms a hyper-sphere containing other vertices, the majority of other vertices should be at a distance greater than $\frac{r}{2}$ from $v_i$. As such we expect a normal distribution of data points to be somewhat right-heavy.

Figure 4.3 shows several cases which demonstrate the intuition of our method. Case A is an example where the distribution of edge lengths for a given vertex is extremely right-heavy. This means the majority of nearest neighbours are similar in distance to the farthest neighbour. In this case it is reasonable to assume that further nearest neighbours would be of a similar distance, and thus increasing the $K$ connectivity would not be a problem. This kind of distribution would imply that the farthest

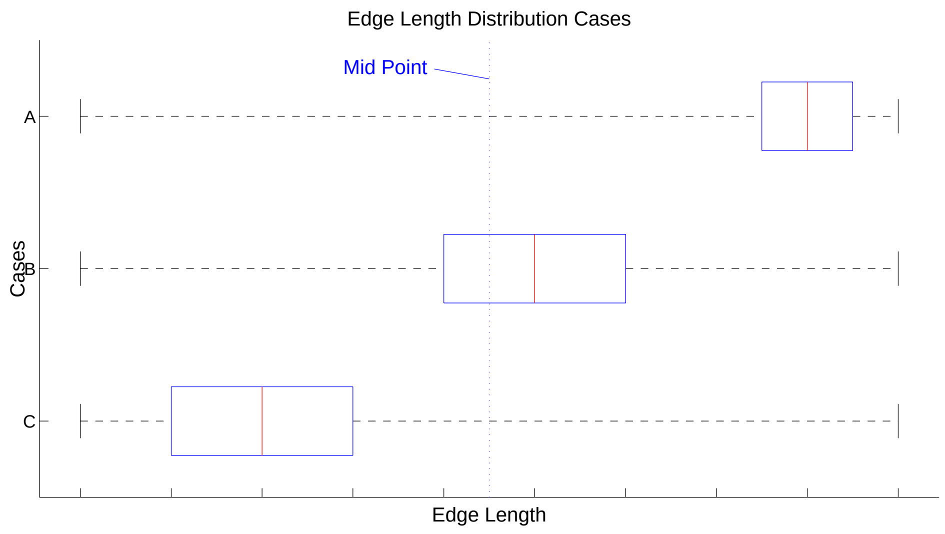


Figure 4.3: Different cases for the distribution of edge lengths.

neighbours are more likely to be intra-class edges, which are good for our purposes. Case A would result in a very high distribution score, as per our definition of it, since the distance of the median from the midpoint is significantly higher than the IQR. This is particularly the case in extremely low $K$ values. To calculate this we need a connectivity of at least 4, or more, outgoing edges, due to its reliance on median and inter quartile range, and thus this is the lower-limit. With such a small number of data points the edge length variance is likely to be extremely high, which will result in a very high edge distribution score.

Case B in Figure 4.3 shows an example of a somewhat right-heavy distribution of edge lengths. In this case the score would be greater than 0, but less than 1 because the median is not significantly higher than the midpoint of the data, with respect to the IQR. In this case, increasing the $K$ connectivity and adding additional neighbours would further push the median towards the midpoint, which, as we established, is not to be expected in an even distribution of data points. Adding additional neighbours when the distribution of edge lengths is like this will result in additional edges which are a significant distance from the centre of the edge length distribution, and therefore are more likely to be inter-class edges due to their significantly longer resulting edges compared to a majority of the other existing edges. In this case we prefer to lower the $K$ connectivity rather than increase it, which will discard points in the farthest quartile, such that the data is likely to become more right-heavy.

In the final Case C in Figure 4.3 we see a left-heavy distribution. According to our definition, a left-heavy distribution like this will always result in a score less than zero. If the centre median of the distribution of edge lengths is less than our midpoint then that means the farthest neighbours are significantly more distant from the vertex than the majority of nearest neighbours. In this case, the large variation in edge lengths connected to the farthest neighbours means that these are more likely to be inter-class edges, which we wish to avoid. This case would result in a score below zero, and would mean we wish to decrease our $K$ value, to discard the farthest edges which are more likely to be inter-class edges.

As we can tell from these three cases, in general a higher distribution score means we wish to increase our $K$ connectivity parameter, and a lower distribution score means we with to decrease our $K$ value. In the next section we will show exactly when we intend to change the $K$ value according to the distribution score.

### 4.2.3 Selection of the $K$ Parameter

Our parameter selection method uses the edge distribution score in varying the $K$ value over time. The intuition behind edge distribution score is that the distribution of the edge lengths is relatively predictable as $K$ increases, or else the $K$ connectivity parameter should be decreased. Since this score represents how far the distribution of edge lengths deviates from what we expect in an even distribution of data points, there will be some $K$ value at which there are a large number of inter-cluster edges, which result in an edge distribution score different than what we expect. Similarly when the distribution of edge lengths is indicative of having too low a $K$ value, the score should reflect that. As such, to select the appropriate $K$ value, we must just select based on the value of $K$ that produces a score close to what we expect the distribution to be. This comes in the form of a constant threshold, which we describe below.

The edge distribution score is calculated for all data points, represented by vertices in a sparse graph, in memory. The individual score for each vertex is calculated on is $K$-nearest neighbours on the point-level sparse graph, and then the mean value of those distribution scores is used. This is referred to as the *average distribution score* (ADS). The ADS is used when we determine when to adjust the $K$ value in our method.

We define a constant threshold $\theta = 2.0$, which means that on average the distribution of edge-lengths should be right-heavy, as we describe previously, and ideally be around one inter-quartile range from the midpoint. Our edge distribution score is calculated as the distance of the midpoint from the median in relation to half of the IQR, and so a $\theta$ of 2.0 represents a distance exactly equal to the IQR - that is, when the edge distribution score of a vertex is exactly 2.0, then the distance from the median

$m_i$ to the midpoint $\mu_i$ will be exactly equal to the interquartile range $r_i$. When a vertex produces a score greater than this constant it will indicate that a higher $K$ value is required, whereas if the median length is less than the midpoint distance then the score will be below the threshold, which is a sign the $K$ value might need to be decreased. The threshold constant of $\theta = 2.0$ allows us to maintain an expected right-heavy distribution, whilst also reducing the amount of potential inter-class edges.

**Algorithm 4** Dynamic $K$ selection algorithm for RepStream. K selection occurs periodically before new vertices are linked into the RepStream sparse graph

```
 1: $FUNCTION : RepStreamKSelection$
 2: $INPUT : X, K, M$ {$X$ is the set of data points in the stream, $K$ is the initial con-
    nectivity value, and $M$ is the maximum number of points to store in memory}

 3: $\theta \leftarrow 2.0$
 4: $margin \leftarrow 0.2$
 5: for all $x_i$ in $X$ do
 6:     if $i$ is evenly divisible by $\frac{M}{10}$ then
 7:         $Score \leftarrow$ Compute ADS for all vertices in memory
 8:         if $Score > \theta + margin$ then
 9:             $K \leftarrow K + 1$
10:         else if $Score < \theta - margin$ then
11:             $K \leftarrow K - 1$
12:         end if
13:     end if
14: end for
15: $Neighbours \leftarrow$ K nearest neighbours of data point $x_i$
16: Data point $x_i$ becomes a vertex $v_i$
17: $LinkIntoGraphSG(v_i)$
```

The algorithm for the dynamic selection method for $K$ is shown in Algorithm 4. In this algorithm the inputs are $X$, the set of all data points in the stream $< x_1, x_2, \ldots, x_{|s|} >$, $K$ is the initial $K$ connectivity value which our algorithm will adjust over time, and $M$ is the maximum number of points that RepStream is to store in memory. The symbol $\theta$ denotes our threshold constant, which we define as $\theta = 2.0$, and *margin* which allows for a margin of error in the distribution score. Our algorithm allows RepStream to work as normal, but periodically computes the average distribution score, ADS, for each vertex currently in RepStream's point level sparse graph. The $K$ value is adjusted up or down by a value of 1, or kept the same, depending on what the average distribution score is computed as. By incrementing or decrementing the $K$ value by one each time, the algorithm can converge on the correct solution steadily over time.

The $K$ adjustment process takes place every $\frac{M}{10}$ data points, which allows time for the change in $K$ to stabilise before the score is re-computed.

Our reasoning for including a *margin* is that the value of the distribution score is exceedingly unlikely to exactly equal the threshold parameter, and so it is likely that even at the correct level of connectivity $K$ will alternate between two values repeatedly by fluctuating above and below the threshold. Thus a margin is included to allow for more stability in the $K$ values selected, meaning the algorithm is less sensitive to noise, and more likely to respond only to changes in the data distribution over time. We allow a margin of 10% of the threshold constant, meaning for our threshold of $\theta = 2.0$ there will be no changes if the calculated ADS is between 1.8 and 2.2. For our method we must set an initial $K$ value which is then adjusted over time by our distribution score selection method, as we describe in Section 4.3 we attempt our best to select the *least optimal* initial $K$ value possible in order to show how our method works under worst-case conditions.

## 4.3 Evaluation

### 4.3.1 Synthetic Datasets

We begin with a selection of synthetic datasets , which are designed to present difficult clustering problems. The datasets, aside from DS1 and DS2, are datasets that evolve over time - with distributions that move, change size, or change density during different points of the data stream. While synthetic datasets are not necessarily representative of typical real-world datasets, they can be crafted such that they provide specific challenges which are difficult for clustering algorithms to deal with.

**DS1 and DS2** are synthetic datasets used in the original RepStream paper (Lühr & Lazarescu, 2009) which contains static distributions of data which pose difficult challenges for clustering algorithms to deal with. They can be seen in Figures 4.4 and 4.5, which shows how the distribution contains classes with concave shapes, as well as classes which are contained within other classes. Synthetic datasets like this are a commonly used way to test the general effectiveness of a clustering algorithm on a static dataset.

**SynTest** dataset is an evolving dataset that consists of one persistent class that slowly shifts its shape and position over the course of the stream, as well as several other

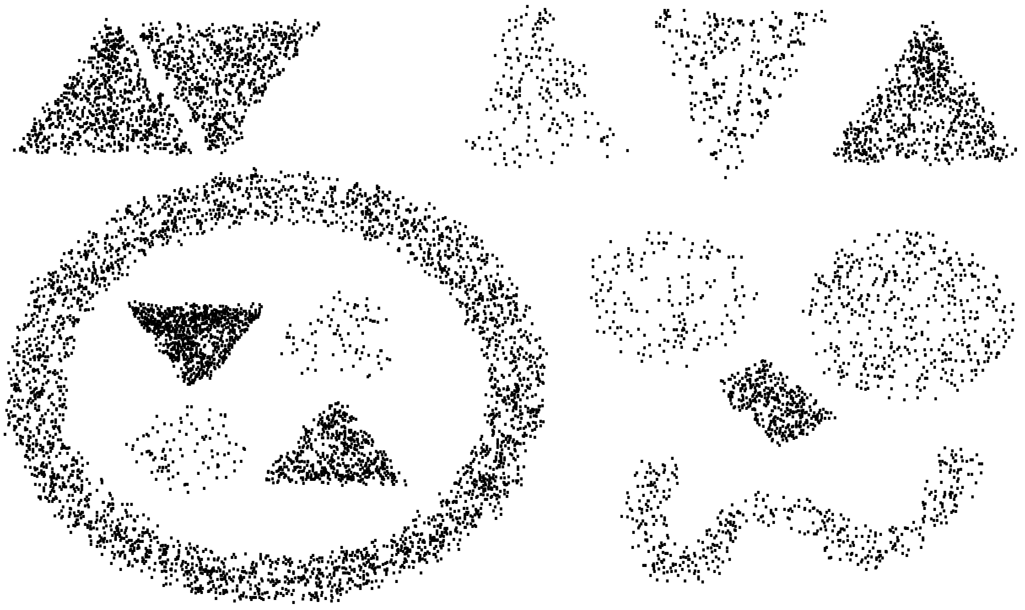


Figure 4.4: Visualisation of the DS1 dataset.

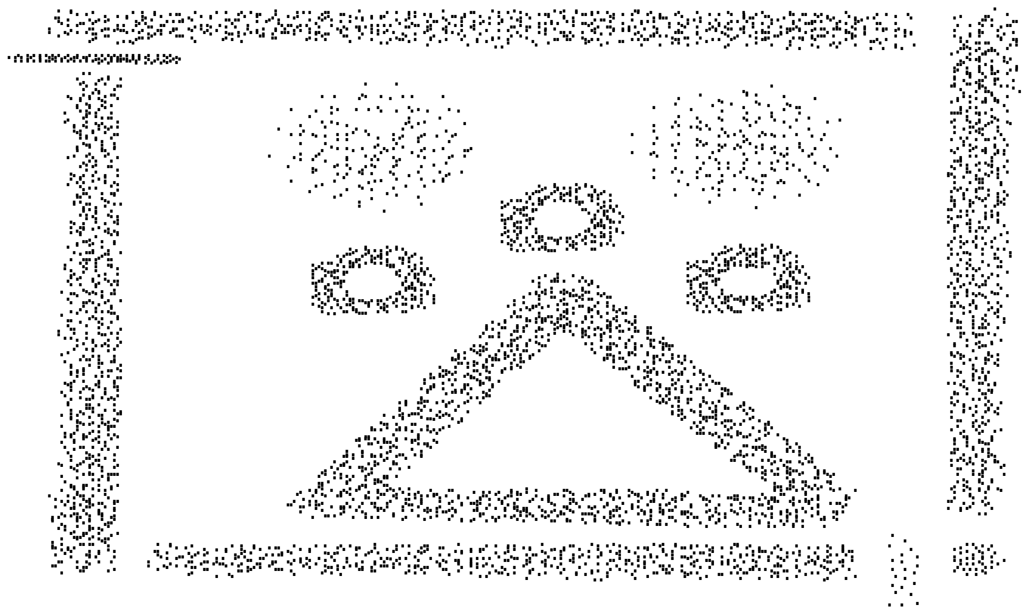


Figure 4.5: Visualisation of the DS2 dataset.

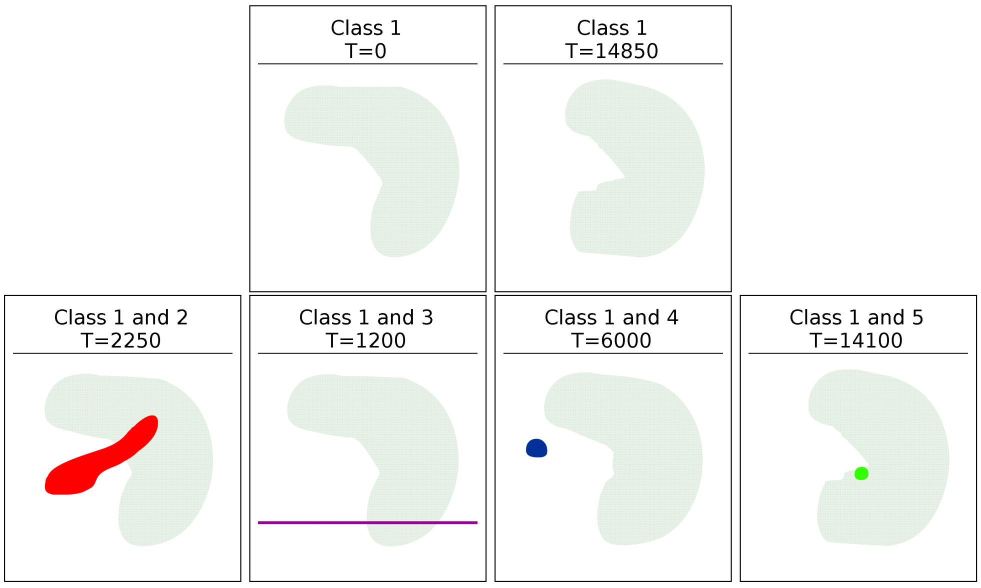


Figure 4.6: Two dimensional representations of the 5 classes in the SynTest dataset. The main class is always present and steadily changes shape, the smaller classes appear at various points through the dataset, as shown in Figure 4.7.

smaller, more dense classes which are transient - appearing and disappearing at various periods of the stream. The larger class is present throughout the whole dataset, and makes up the majority of the data points, while the smaller classes exist for a relatively shorter amount of time. Each of these smaller classes are more dense than the main class, but are present for only a few hundred, to a few thousand time-steps at a time.

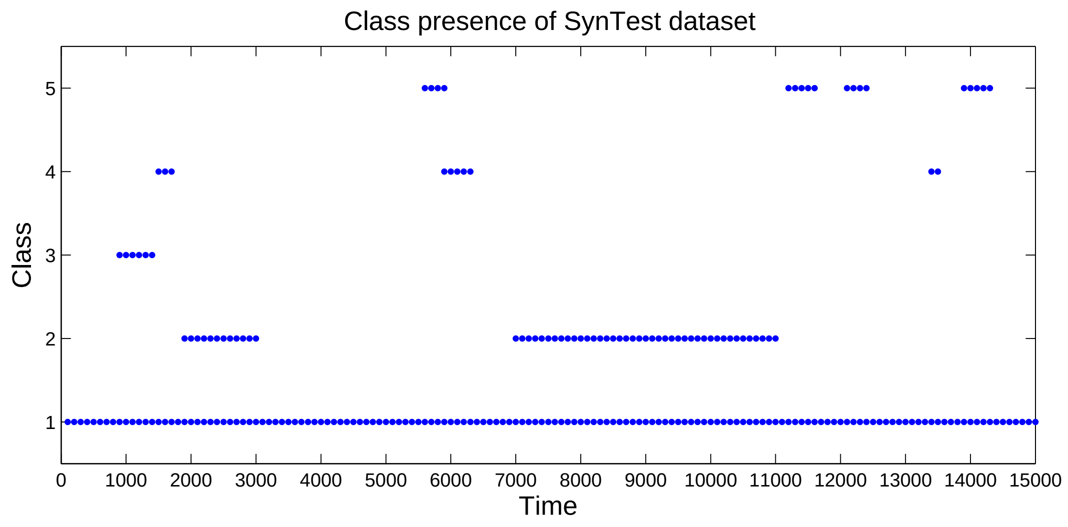


Figure 4.7: The class presence of the classes in the SynTest dataset. A marker indicates that the class is present in the dataset during the given time window.

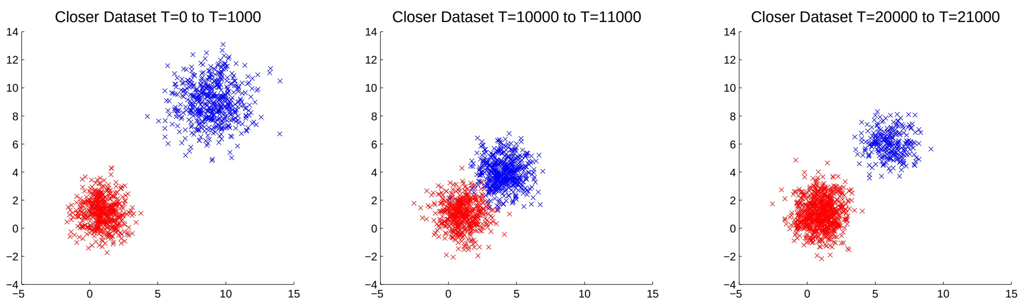


Figure 4.8: The evolution of the Closer dataset, showing slices of its 3 sections.

Figure 4.7 shows the presence of the classes in the SynTest dataset. Adjacent marks indicate when the given class is present in the given time window. The shape, size, and position of the classes is shown in figure 4.6. Class 1 is always present through the dataset, while the other classes are present for shorter time periods.

**Closer** is a dataset which simulates the separation between classes in a dataset becoming smaller over time. There are three distinct stages to the dataset: The first 10,000 data points alternate between two classes, all data points are on a two dimensional plane and each class is normally distributed with a significant level of separation between the two classes. In the next 10,000 data points, the two classes suddenly become much closer together, such that the two classes borders are slightly overlapping. The final 10,000 points return the classes to having a greater degree of separation once again, however one of the classes becomes more dense, while the other becomes less dense. Figure 4.8 shows these three stages. The changes in this dataset are sudden, conforming to the three distinct stages of the dataset.

The three stages of the dataset were sampled like so:

- Between $T = 0$ to $T = 10,000$ class A was centred at 1,1 and normally distributed with a $\sigma$ of 1 in both the x and y axes. Class B was centred at 8,8 with a standard deviation $\sigma$ of 1.5 on both axes. Points were sampled from these distributions alternately between classes.

- Between $T = 10,000$ to $T = 20,000$ class A remained the same, while class B was moved to 4,4. Points were sampled from these distributions alternately between classes.

- Between $T = 20,000$ to $T = 30,000$ class A remained the same, while class B was moved to 6,6 with a standard deviation $\sigma$ of 1.5 on both axes. Points were sampled from these distributions randomly by alternately selecting 3 points from class A, followed by 1 point from class B.

### 4.3.2 Benchmark Datasets

The real-world benchmark datasets described here are examples of a typical usage of stream clustering algorithms. They contain data taken from real-world sources which has been manually labelled such that external validation metrics can be used to determine the effectiveness of algorithms working on them. Because class labels are available we are able to calculate measures such as Purity, Entropy, $F$-Measure, as well as other scores to show the performance of algorithms in an objective sense. Typically, clustering is itself an exploratory process, and class labels and labelled training data is unavailable, thus an algorithm should ideally perform well on benchmark datasets to be trusted to perform well in real-world situations.

**The KDD Cup 1999 dataset** The KDD'99 dataset (Hettich & Bay, 1999) is a well-known benchmark dataset. It is extracted from logs taken from a smart firewall in a network being subjected to simulated and controlled network attacks. It contains high dimensional data, of which we use the 34 numerical features with each data point presented as a 34-dimensional vector. We use a sub-sampled version of the dataset containing 494,020 data points, which is about 10% of the original KDD Cup 1999 dataset. Most of the data in the sub-sampled version of the dataset falls into either the normal traffic class, or one of two major denial-of-service attack classes. A relatively small percentage of the data - less than 2% - are from 20 other network attack types. Each data point is labelled with the type of traffic (normal, or the type of attack) for evaluation purposes.

This KDD Cup 1999 dataset has been used previously in evaluating stream clustering algorithms (Ackermann et al., 2012; Aggarwal et al., 2004; Bhatnagar et al., 2014; Lühr & Lazarescu, 2009) due to the high variability between classes in the dataset. The various network attacks interrupting the normal traffic represent changes in the distribution of subsequent data points, known as concept drift. This is a significant challenge for clustering algorithms to deal with, making it an excellent dataset for testing how an algorithm deals with dynamic, unpredictable data point distributions over time.

**The Tree Cover Type dataset** The Tree Cover dataset (Blackard & Dean, 1999) is a real-world data stream of a set of features extracted from satellite photos and geological surveys from forested areas of northern Colorado. It contains over 580,000 entries with ground truth labels corresponding to which type of trees grow in each area, and has been previously used as a benchmark dataset for stream clustering (Lühr & Lazarescu, 2009; Bhatnagar et al., 2014; Forestiero et al., 2013). This data represents a naturally evolving stream of data which changes with the environment and climate of each region.

A particularly challenging feature of the Tree Cover dataset is that the classes overlap to some degree in some of the dimensions. This makes clustering particularly challenging as overlapping classes means there's no spatial separation which can be used to determine where the edges of the classes are. Instead, changes in the density of the data must be used to find where the different classes lie. Since this is a particularly challenging case it is common that even modern, sophisticated clustering methods have a high error rate in separating data.

### 4.3.3 Experimental Set-Up

To evaluate our method's efficacy we will examine external validation metrics, specifically the purity and the $F$-measure scores. External validation metrics are commonly used (Kaur et al., 2015; Kremer et al., 2011) to evaluate the performance of clustering methods in an objective sense against the stated *ideal* clustering, which is represented by labels for each data point, showing which belong together and which should belong in different clusters.

Our experimental set-up and parametrisation is as follows for our proposed method:

- Memory parameter set to $M = 1000$, as a maximum number of data points in memory at any time.
- The $\alpha$ scaling factor is set to $\alpha = 1.5$ as suggested in the original paper.
- Vanilla normalisation is enabled.
- Decay parameter $\lambda$ at the default value of $\lambda = 0.99$.
- The initial $K$ value is set to the worst possible $K$ value for the dataset, as described below.

Table 4.2: Best and Worst $K$ values for RepStream, according to $F$ score.

| **Dataset** | **Best K** | **Best $F$ score** | **Worst K** | **Worst $F$ score** |
|---|---|---|---|---|
| DS1 | 7 | 0.7208 | 18 | 0.2767 |
| DS2 | 7 | 0.6371 | 21 | 0.2594 |
| SynTest | 9 | 0.8614 | 5 | 0.5435 |
| Closer | 9 | 0.7989 | 5 | 0.4345 |
| TreeCov | 29 | 0.6108 | 5 | 0.2978 |
| KDD99 | 30 | 0.7898 | 5 | 0.2636 |

We use these same values for all datasets, aside from varying the $K$ value.

As mentioned, we set the initial $K$ values for our proposed method to be the worst possible $K$ value we can select. We do this in order to show that even under the worst case scenario our method can still adapt and produce useful results. To determine which $K$ value is the worst we have run the original RepStream multiple times on each dataset for a range of $K$ values between $K = 5$ and $K = 30$. We then use the class labels to determine which $K$ value produces the lowest mean $F$-measure score for each dataset, and which produces the highest mean $F$-measure score. We set a lower floor for the $K$ value of our algorithm to $K$ values between $K = 5$ and $K = 30$ because our method requires a minimum number of outgoing edges to calculate a median and IQR, and because $K$ values higher than 30 are too high to produce distinct clusters in most cases.

Table 4.2 shows the $F$-measure scores produced by running the original RepStream algorithm at different $K$ values, using the same parameters listed above. For our experiments on our own algorithm we use the **worst** initial $K$ values for each dataset as our initial $K$. As can be seen in the table, the overall $F$-measure score can vary a great deal when a sub-optimal $K$ is selected, especially in the worst case scenario. It is worth noting that selecting a $K$ value is not a trivial task, and that since usually one has no access to class labels it is impossible to know whether an appropriate $K$ has been selected.

### 4.3.4 Results vs Other Algorithms

We examine the results for HPStream (Aggarwal et al., 2004), CluStream (Aggarwal et al., 2003), ExCC (Bhatnagar et al., 2014), and STRAP (Zhang et al., 2008), using their published results to compare against our own. Rather than listing purity values at every time in the stream, these papers instead publish purity values for specific time slices. We evaluate our algorithm throughout the entirety of the datasets and record the

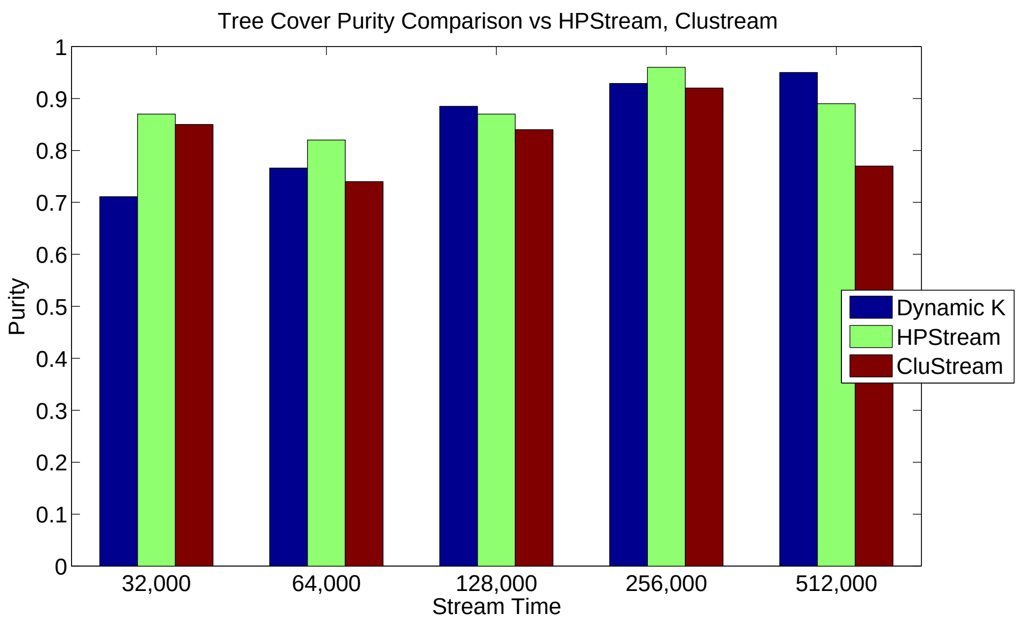


Figure 4.9: Comparative purity for Tree Cover dataset

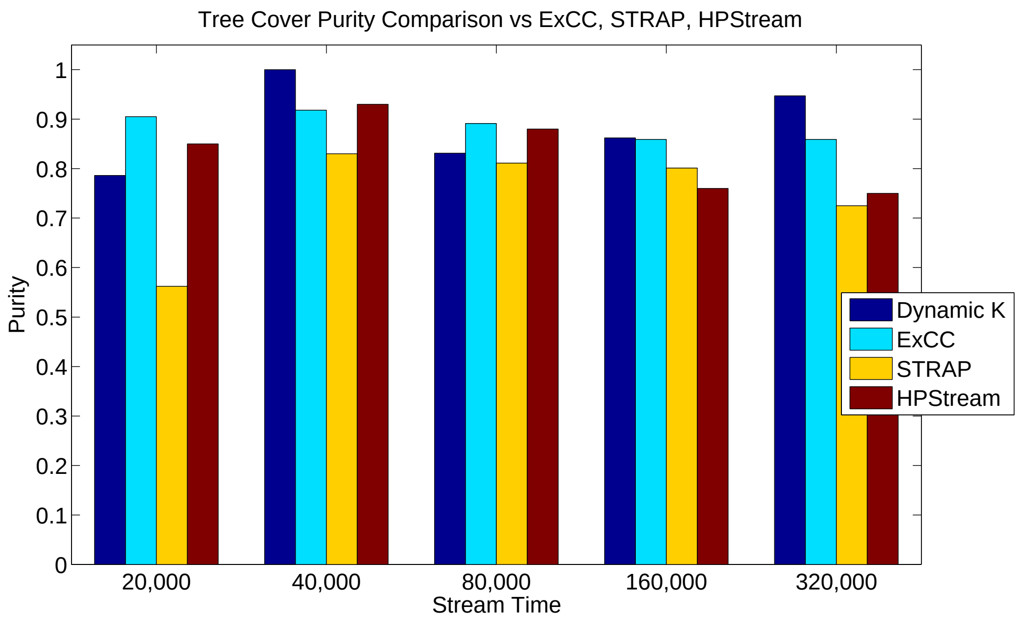


Figure 4.10: Comparative purity for Tree Cover dataset

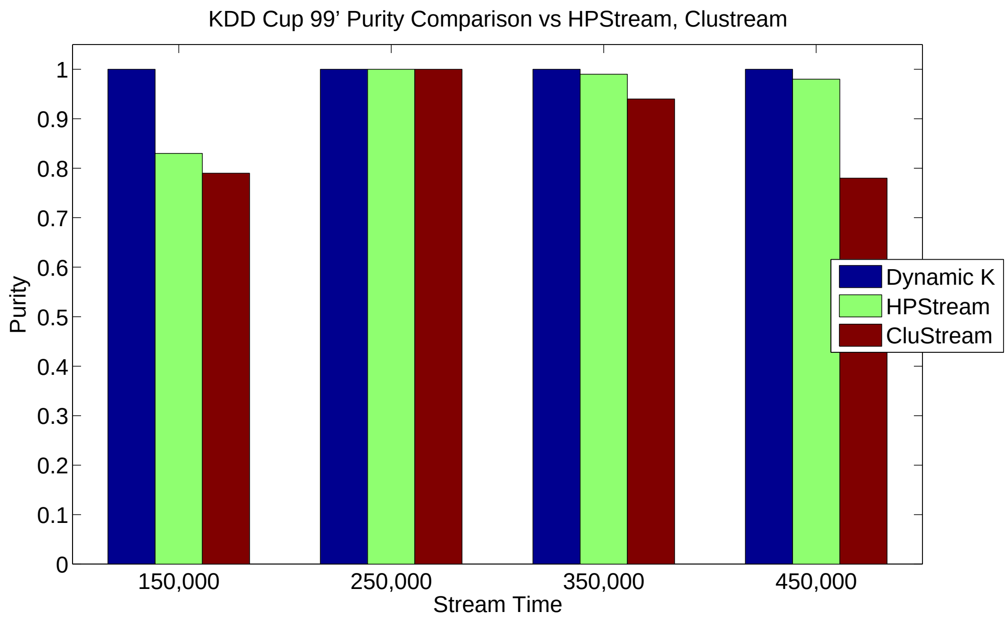


Figure 4.11: Comparative purity for KDD 99' Cup dataset

purity values at these same time slices for comparison. These time slices correspond to attacks in the KDD dataset, times where it is most difficult for clustering to occur. We use purity as the chosen measure because, as a common external validation metric, it is the measure used in the published results for these algorithms. We also use the Manhattan distance metric for these evaluations, as it is efficient to compute.

Tree Cover Type dataset, Figure 4.11 shows the comparisons against our method (Dynamic $K$) HPStream and CluStream, while Figure 4.12 shows the comparisons between our method, ExCC, STRAP, and HPStream. As we can see, all algorithms are able to achieve a greater than 0.7 purity for each time slice, except for STRAP which drops produces a less than 0.6 purity value at the 20,000 time-slice. Our Dynamic $K$ RepStream method performs favourably against the other algorithms, achieving a higher purity than the other algorithms in 5 of the time-slices, and performing similarly well to the other algorithms in the other time-slices. Earlier time-slices struggle with the purity values, however as the algorithm stabilises over time the results become more consistently high, and equal or out-perform the other algorithms. The $K$ value selected by our algorithm is initially set at the worst value of 5, but over time our method increases this value to a more appropriate value, reaching up to the maximum $K$ value of 30 at times.

For the KDD dataset, Figure 4.9 and Figure 4.10 show the comparisons of our method against the same dataset listed previously. HPStream performs well on the

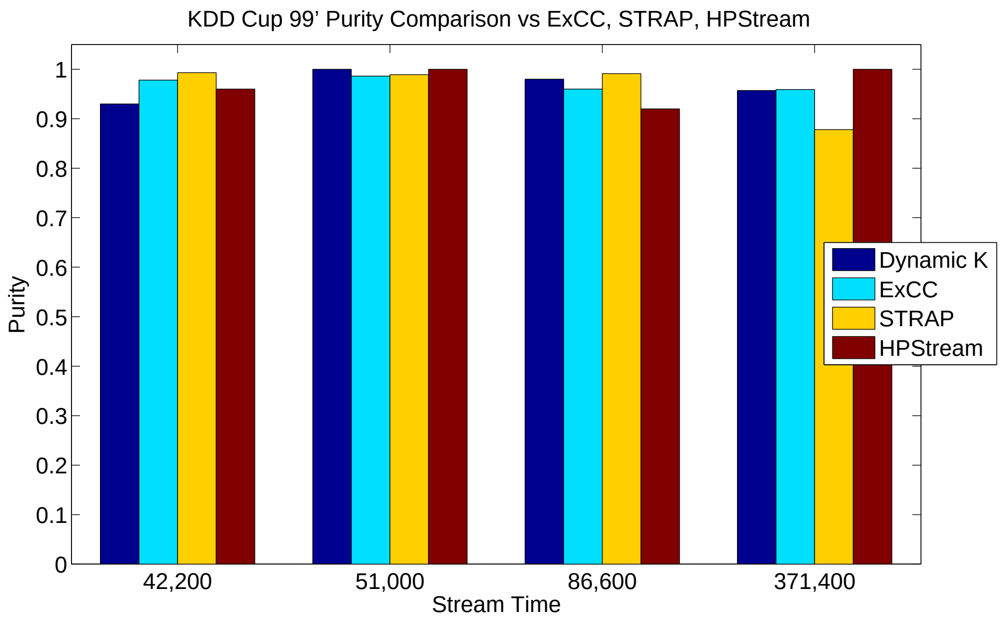


Figure 4.12: Comparative purity for KDD 99' Cup dataset

KDD dataset compared to the other algorithms, which is expected because the algorithm is designed specifically to handle high dimensionality datasets like the KDD dataset. Our extended version of RepStream with the dynamically selected $K$ value, outperforms HPStream and CluStream in all time slices in Figure 4.10, and also outperforms ExCC, HPStream, and STRAP in half of the time slices in Figure 4.9. Our method tends to struggle nearer to the start of the dataset when configured with a poor initial $K$ value, but is able to adjust over time.

Additionally, we also compare against the DBStream (Hahsler & Bolaos, 2016) and D-Stream (Chen & Tu, 2007) algorithms, which had implementations available for the Stream package (Forrest, 2011) of the R programming language. We used the recommended parameters for each of these algorithms according to the original papers for each algorithm. D-Stream grid-size parameter was set to $len = 0.05$, its dense and sparse cell thresholds set to $C_m = 3.0$ and $C_m = 0.8$, the decay value $\lambda = 0.998$, and its sporadic cell deletion parameter $\beta = 0.3$. The DBStream algorithm was set with its micro-cluster radius $r = 0.05$, its decay parameter $\lambda = 0.01$, its clean-up interval $t\_gap = 1000$, the minimum weight $w\_min = 3.0$, and its intersection factor $\alpha = 0.1$. We then ran DBStream and D-Stream on each of our synthetic and benchmark datasets, calculating the purity values at 100-point intervals.

Figure 4.13 and Figure 4.15 show the purity scores calculated for the DS1 and DS2 datasets respectively on the DBStream, D-Stream, and our dynamic $K$ method. In both

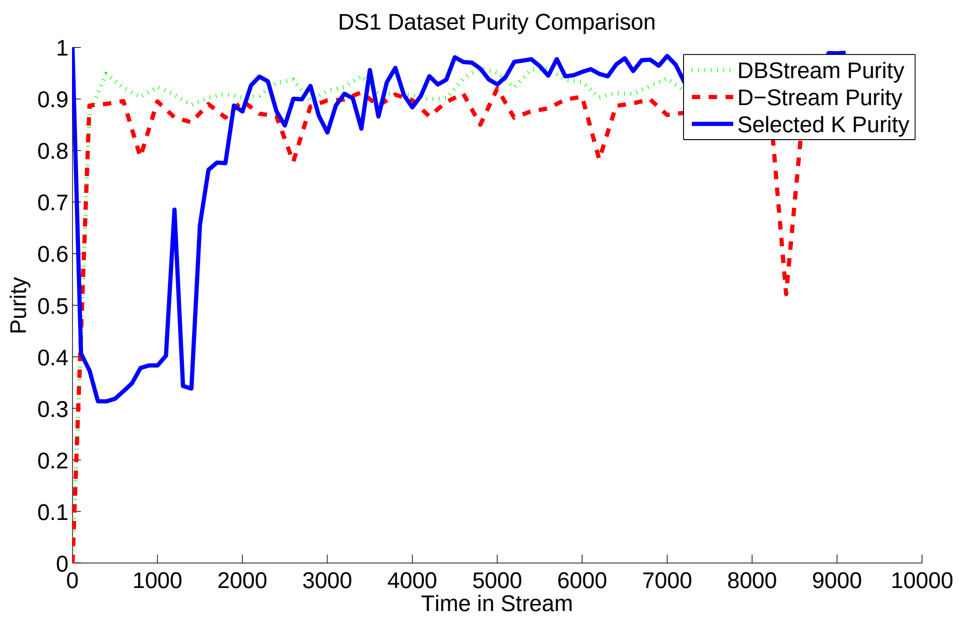


Figure 4.13: Comparative purity for DS1 dataset

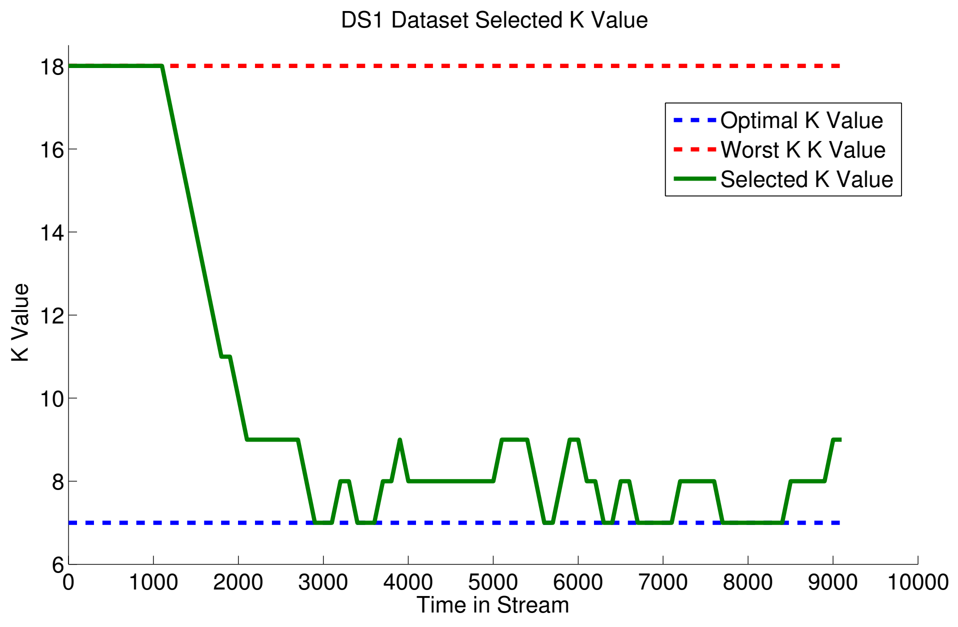


Figure 4.14: The *K* value selected by our dynamic *K* method on the DS1 dataset

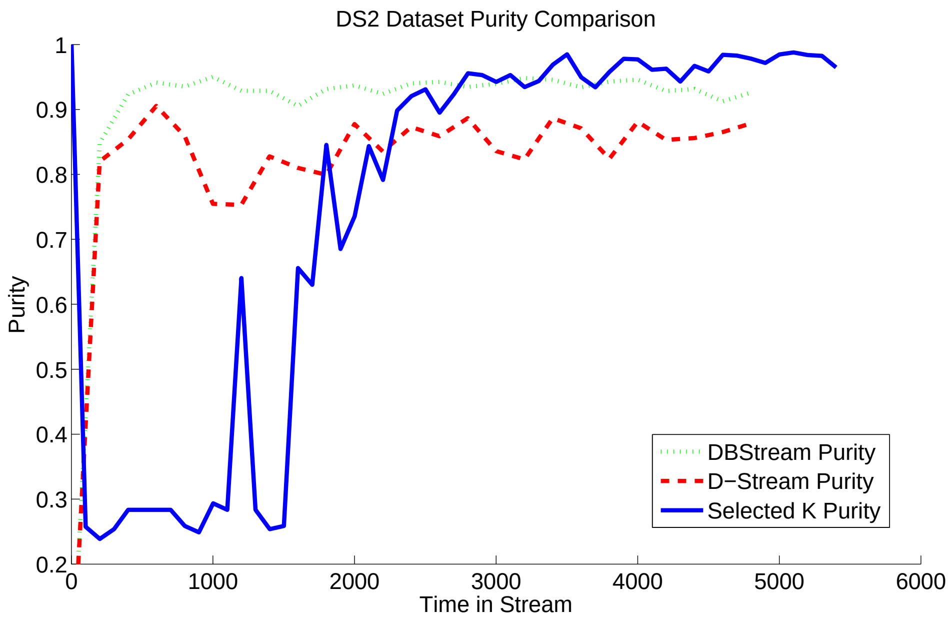


Figure 4.15: Comparative purity for DS2 dataset

of these datasets a similar phenomenon happens - at first D-Stream and DBStream outperform our method significantly for the first approximately 2000 data points, with our method having a purity score hovering as low as 0.3 purity. After this time, however, our algorithm shows a steep increase in clustering purity until, in both cases, our method outperforms the other algorithms. The low initial purity of our method is due to our attempts at selecting the worst possible initial $K$ value as the input parameters (in this case $K = 18$ for DS1 and $K = 21$ for DS2). After some time to adjust the internal $K$ value in response to the calculated edge distribution score the algorithm performs better, and for both datasets our methods adjusts the $K$ value downwards, hovering around $K = 9$, which according to Table 4.2 is close to the optimal average $K$ value.

We show an example of the $K$ value adjusting over time in Figure 4.14. Initially our $K$ parameter begins at the worst possible $K$ value of $K = 18$, but our algorithm very rapidly determines that this $K$ value is too high, and the $K$ value is decreased each time step until it reaches a reasonably stable value after $t = 2000$. The $K$ value doesn't sit exactly on the optimal value, but hovers at a comparable value, which is why the purity, shown in Figure 4.13, increases dramatically after $t = 2000$.

For the SynTest dataset, we see in Figure 4.16 that our method performs comparably well versus the DBStream and D-Stream algorithms. The results are close enough that it is not easy to determine which produces a higher overall purity by looking at the plot alone, so instead we take the mean purity values. D-Stream's mean purity value

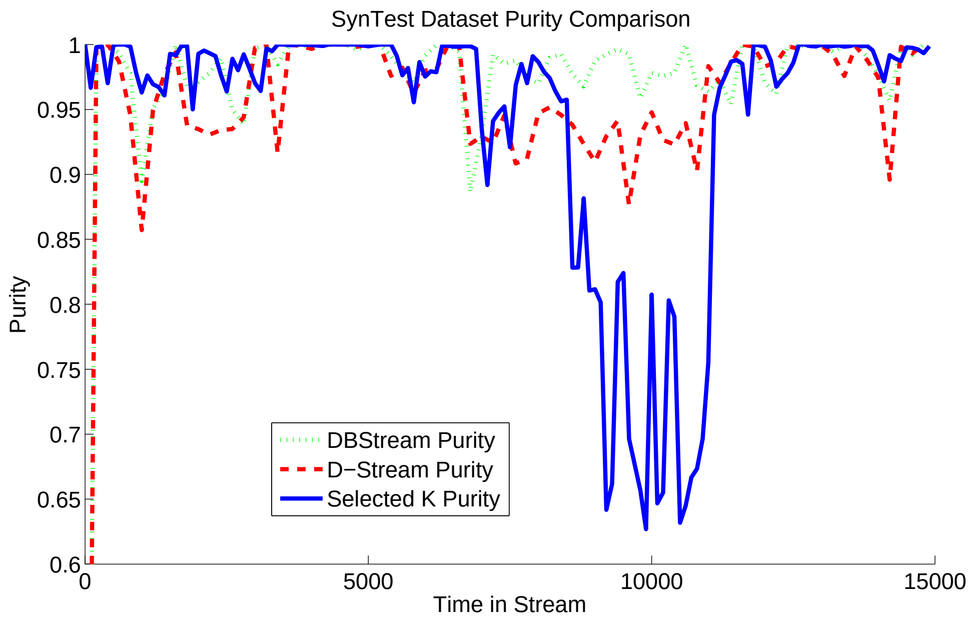


Figure 4.16: Comparative purity for SynTest dataset

is 0.951, our method produces a mean purity value of 0.943, while DBStream's mean purity is 0.969. Overall, none of the methods stand out as being clearly superior for this dataset, in fact all achieve high purity clustering throughout the stream, though our method does dip in performance briefly around the 10,000 mark.

The three algorithms perform very differently from each other on the Closer dataset, shown in Figure 4.17. The DBStream algorithm produces good purity results for the first and last 10,000 data points in the 30,000 point dataset, but performs very poorly from 10,000-20,000, dipping as low as 0.5 purity. D-Stream, on the other hand, has purity results which are remarkably consistent throughout the dataset, including in the middle section where the distribution has two overlapping classes. Our dynamic $K$ method performs exceptionally well for the first and last 10,000 data points, but has variable success during the middle section of the dataset. The purity in this section varies between 0.5 and 0.9, having an average purity of 0.836 during this section. Overall, however our dynamic $K$ method achieves an overall purity of 0.941, compared to 0.913 and 0.799 for D-Stream and DBStream respectively.

As for the benchmark datasets, Figure 4.18 shows the purity plots for the comparison algorithms on the Tree Cover Type dataset. The plot is very noisy due to the high level of variability in clustering results over time for all algorithms, however it does show that the DBStream algorithm has the highest amount of variability in its clustering quality, occasionally producing zero clusters, and therefore achieving a nominal

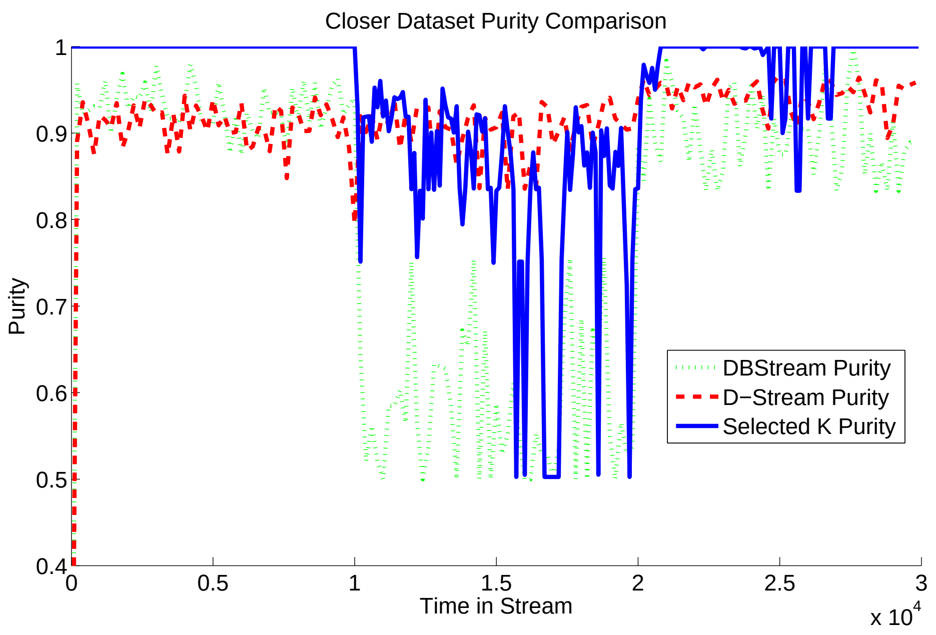


Figure 4.17: Comparative purity for Closer dataset

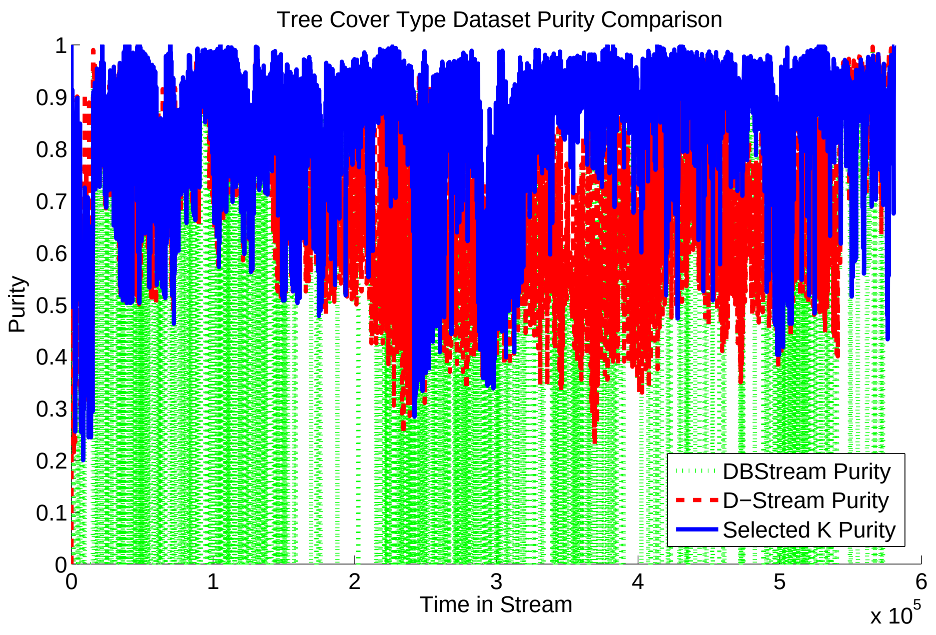


Figure 4.18: Comparative purity for Tree Cover dataset

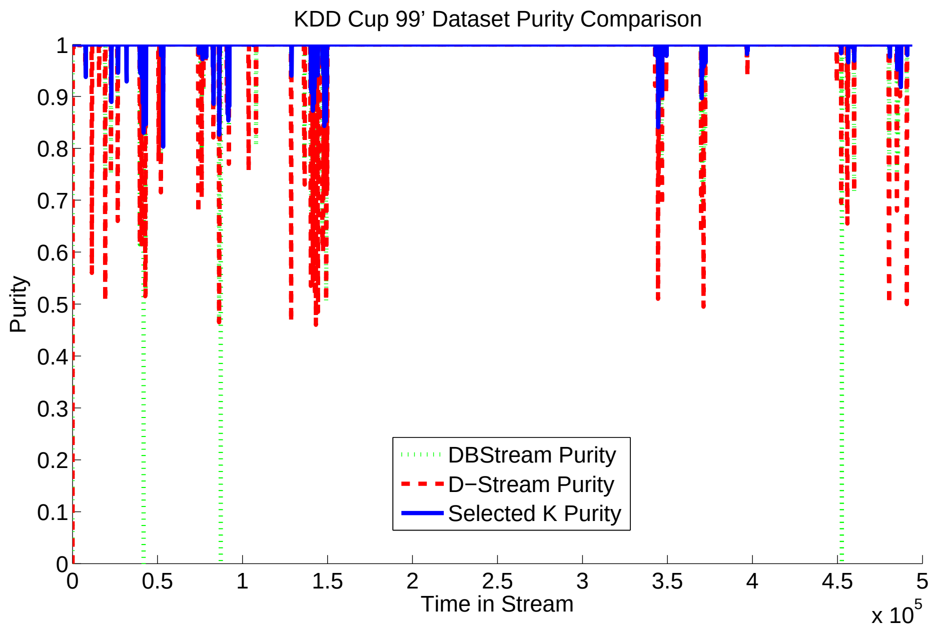


Figure 4.19: Comparative purity for KDD 99' Cup dataset

purity score of 0. At other times this algorithm produces purity values comparable to D-Stream and our dynamic $K$ method. Overall, the DBStream method produces a mean purity of 0.508 (or 0.698 if the zero-cluster instances are discarded), the D-Stream algorithm has a mean purity of 0.690 and our dynamic $K$ method results in a mean purity of 0.843 through the dataset, significantly higher than the other algorithms. While there is still a lot of variability in the results of all algorithms, our method manages to maintain the most consistent and highest purity value on the Tree Cover benchmark dataset. The high degree of overlapping classes in the Tree Cover dataset makes this particularly challenging for algorithms to achieve.

Finally, our method is tested on the KDD Cup 99' network intrusion benchmark dataset, shown in Figure 4.19. Note that very clearly all algorithms sit at 1.0 purity throughout the vast majority of the data stream. This is to be expected due to the composition of the KDD dataset, being made up mostly of normal traffic, and two major attack classes. During most of the time, only one class is present and so any clustering output would result in perfect purity. The sections where there is a decrease in purity corresponds to instances of other classes appearing suddenly in the stream, mixing in with the other traffic. These instances of other classes appearing in the dataset last for a short time before returning to either normal traffic or one of the attack classes. Whilst all algorithms maintain a very high purity overall, the decrease in purity during the short attack instances is smaller in our dynamic $K$ method than in

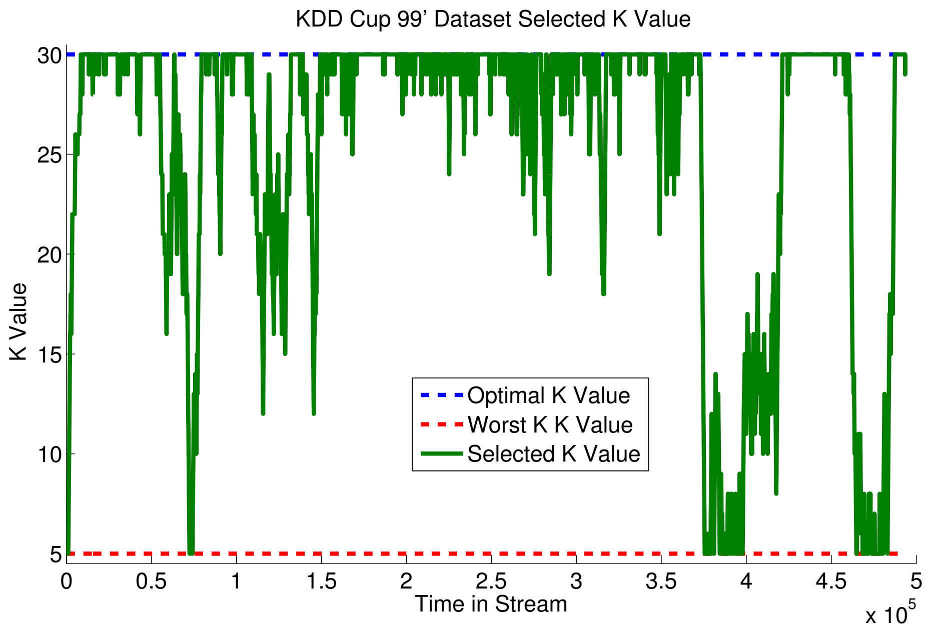


Figure 4.20: The $K$ value selected by our dynamic $K$ method on the KDD Dataset

the DBStream and D-Stream methods. Our method maintains a purity above 0.80 for the whole stream, keeping the majority of the traffic separated from the attack classes more effectively than the comparison algorithms.

Figure 4.20 shows the $K$ value selected by our method over time while being run on the KDD Cup 99' dataset. Initially we set the algorithm to the worst possible initial $K$ value of $K = 5$, but it very rapidly determines that this value is too low, and increases the $K$ value to our cap of $K = 30$. This suggests that it would also tend to go higher if we removed our maximum limit for our evaluations. In general during this dataset the algorithm tends to stay at higher values, between 25 to 30, however it drops down on several occasions, particularly after $t = 3.7 \times 10^5$. Note that while $K = 30$ is the value which provided the highest average purity over the entire length of the dataset, there may be different $K$ values which result in higher purity values at specific times during the stream, as such it is difficult to figure out which is the best exact $K$ value at each time step, especially given that the KDD dataset has a large number of dramatic changes in data distribution over time. The dynamic nature of our algorithm is likely why it performs well compared to standard RepStream, as we show below.

Overall our method performs very well compared to all the algorithms listed above - HPStream, ExCC, CluStream, STRAP, DBStream, and D-Stream. We note that our method was consistently configured such that the initial $K$ value parameter was the worst case possible, while our comparison algorithms were using published results, and parameter values suggested in their original papers. Despite our method having worst-case parametrisation, it was still able to produce comparable results, and even exceed the performance of other stream clustering algorithms after being given time to adjusts its internal $K$ parameter.

### 4.3.5 Results vs RepStream

We wish to also evaluate our method against the ideal results which can be produced by RepStream in an ideal case. As such, we run RepStream using the ideal optimal $K$ value, and compare it against our method. Again, we run our dynamic $K$ selection by selecting the most sub-optimal initial $K$ value and let the value automatically change over time according to the computed distribution score.

For these evaluations we use $F$-measure as our external validation metric. One major problem with purity , as noted in (Kaur et al., 2015), is that purity is an unreliable indicator of performance, despite its popularity as a comparison tool between clustering algorithms. Purity has the problem has no penalty for producing too many clusters, and splitting classes up into multiple clusters. Notably, it is possible to reach a perfect purity value when each data point is treated as a separate cluster. Instead we choose to use the $F$-measure score to compare our results, as this avoids this problem, rewarding both the precision and the recall of the clustering output, rather than just the precision as with the purity value.

Figure 4.21 and Figure 4.22 show the $F$-measure achieved by RepStream configured with the optimal $K$ value against our dynamic $K$ method set with the worst possible initial $K$ value. As one would expect, optimally configured RepStream performs better than our dataset, after about 2000 data points our method catches up in $F$-measure score dramatically. Our method adjusts its internal $K$ value by $\pm 1$ for every 100 data point, and so it takes a period of time for it to achieve a stable $K$ value if it is initially very far off from where it should be. As such in this case it takes approximately 2000-3000 data points before its performance is able to adjust. Our method shows a dramatic upturn in the $F$-measure score after this time period, despite having the worst possible initial setting. Once the internal $K$ parameter stabilises the clustering quality is comparable to that of the optimally configured RepStream. Also marked on Figures 4.21 and 4.22 is the $F$-measure produced by RepStream configured at the worst possible single $K$ value for comparison purposes.

Figure 4.23 has our dynamic $K$ method compare against RepStream set to the optimal value of $K = 9$, and also compared to the worst possible $K$ value of $K = 5$. As

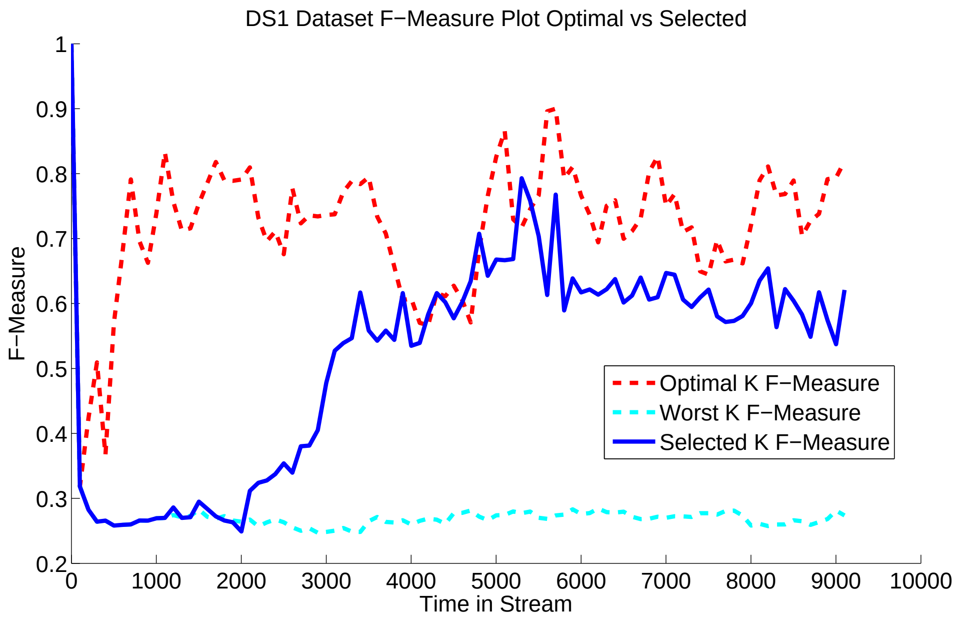


Figure 4.21: F-Measure comparison vs RepStream using optimal parameters on DS1 dataset

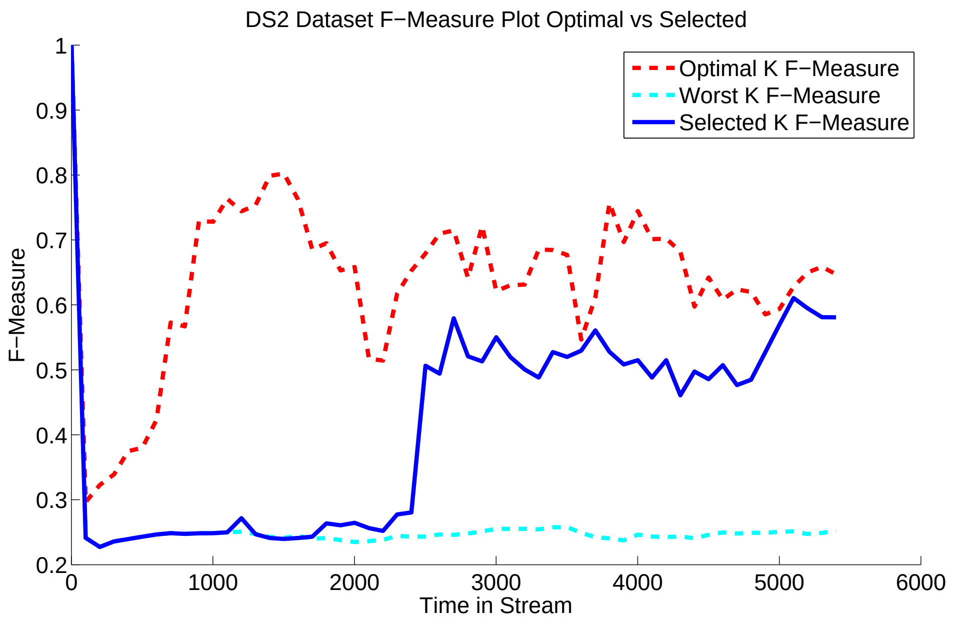


Figure 4.22: F-Measure comparison vs RepStream using optimal parameters on DS2 dataset

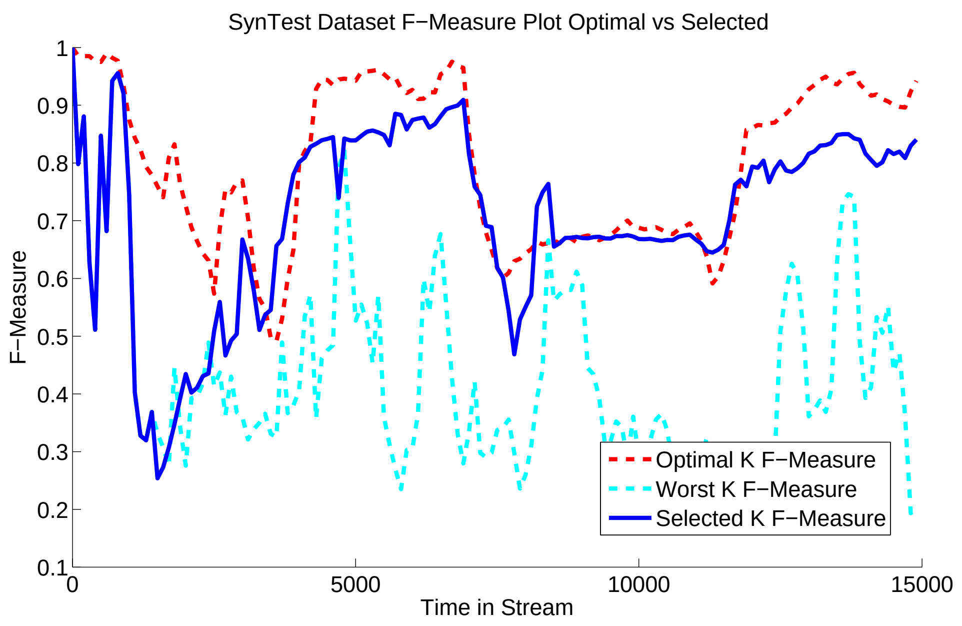


Figure 4.23: F-Measure comparison vs RepStream using optimal parameters on SynTest dataset

expected, our method begins sub-optimally, and over the first 2000 data points compares very poorly to the optimally configured RepStream instance. however, after this time period the $F$-measure score produced by our method dramatically improves, often matching the score of our benchmark. There is a slight decrease in $F$-measure compared to the optimal in the 7000 range, but this difference is only about 0.1 in score versus the optimal.

The Closer dataset is shown in Figure 4.24. RepStream in this experiment has the $K$ value set to $K = 9$ while our dynamic $K$ method has an initial $K$ value of $K = 5$, which the worst initial $K$ value according to Table 4.2. Again, for the first 2000 data points the difference in $F$-measure values is very noticeable, but after this time our method adjusts and almost entirely matches the optimally configured RepStream instances. Notably, there are brief times when our method even outperforms the base RepStream algorithm. On average though, our method has almost identical performance to the RepStream algorithm over most of the dataset having an overall $F$-measure score of 0.841 against the optimal RepStream overall score of 0.861, differing by only 0.02, even when taking into account the initial 2000 data points of poor performance as our method adjusts.

The results for the Tree Cover Type benchmark dataset is shown in Figure 4.25. This dataset is particularly difficult to cluster as it contains overlapping classes. Overall

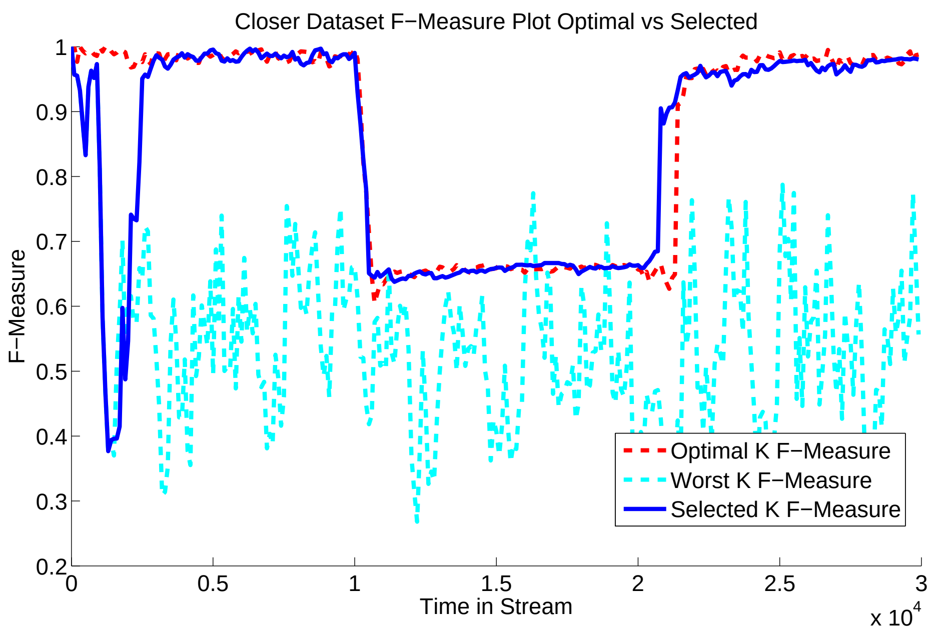


Figure 4.24: F-Measure comparison vs RepStream using optimal parameters on Closer dataset

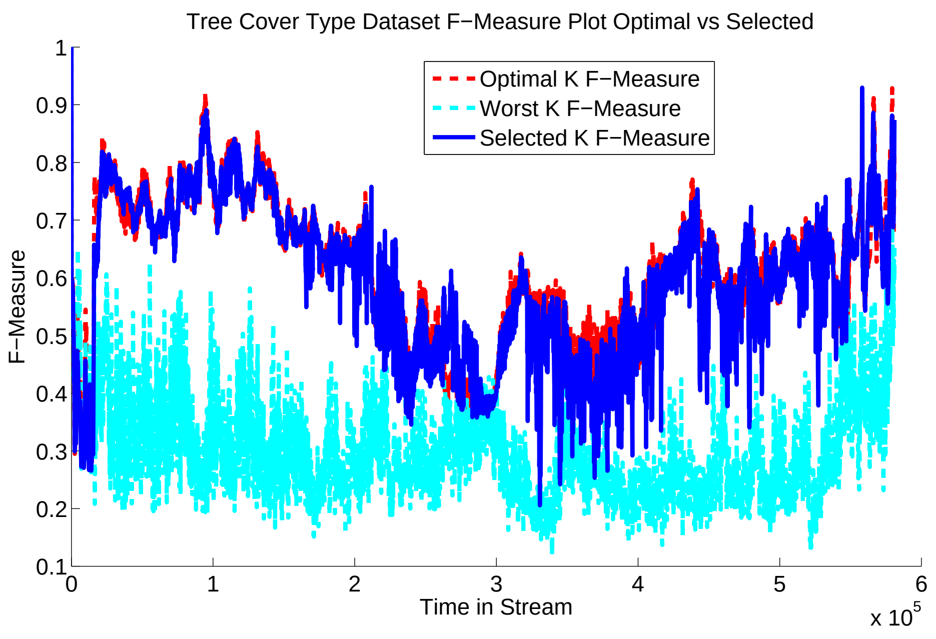


Figure 4.25: F-Measure comparison vs RepStream using optimal parameters on Tree Cover dataset

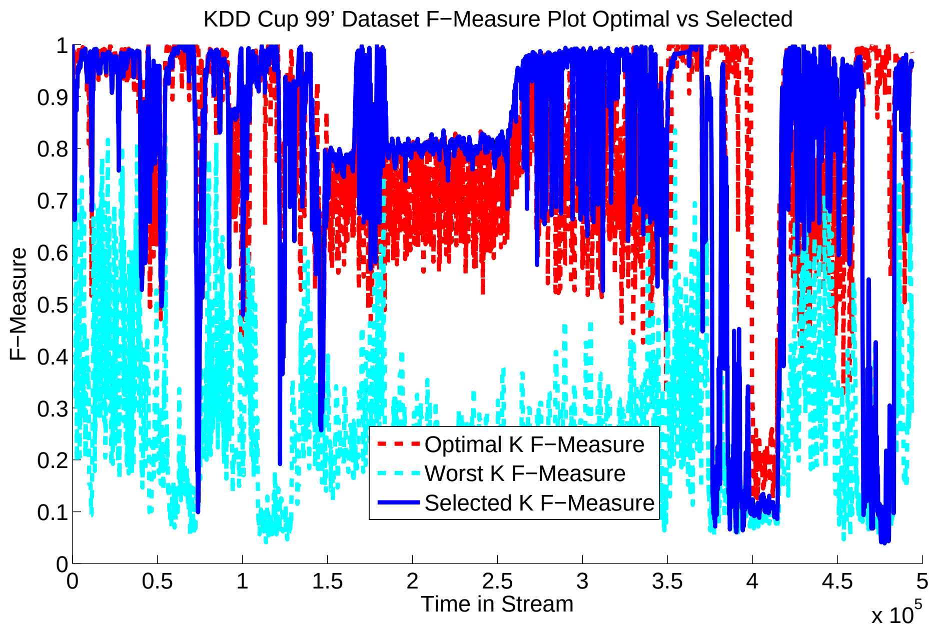


Figure 4.26: F-Measure comparison vs RepStream using optimal parameters on KDD 99' Cup dataset

clustering purity is likely to be imperfect because of this. As such, even the optimally configured RepStream instance set to $K = 29$ has an overall $F$-measure value of 0.611. This is, however, not far off from our dynamic $K$ method which has an overall average score of 0.592, a difference of only around 0.019 overall. As shown in Table 4.2 the average $F$-measure score produced by RepStream if it is kept at the worst possible $K$ value is 0.298, so our method is able to perform exceptionally well even in the face of having the worst initial parameters.

Similar is true with the KDD Cup 99' benchmark dataset. The main challenge in clustering this dataset is combining all normal traffic, and all denial-of-service attack classes into single clusters whilst also successfully differentiating the other attack classes from each other. Thus, for this non-typical dataset the way to produce the best overall $F$-measure score is to combine as many points together into the same cluster as possible. This is why the highest $K$ value we tested for RepStream was optimal. Our method overall performs within 0.1 $F$-measure value of the optimal $K$, producing 0.788 $F$-measure compared to the 0.790 purity of the optimally configured RepStream. This score is vastly improved from the 0.264 $F$-measure score which would've been produced with standard RepStream set to the same initial $K$ value as our method was set to, which demonstrates how much our dynamic $K$ method can help in cases of poor initial parametrisation. As is evident in the plot there are time periods where our

Table 4.3: Comparison of our dynamic $K$ method versus RepStream with optimal and worst $K$ values.

| **Dataset** | **Best F-Measure** | **Worst F-Measure** | **Dynamic $K$** |
|---|---|---|---|
| DS1 | 0.7208 | 0.2767 | 0.5153 |
| DS2 | 0.6371 | 0.2594 | 0.4137 |
| SynTest | 0.7989 | 0.5435 | 0.7091 |
| Closer | 0.8614 | 0.4345 | 0.8410 |
| TreeCov | 0.6108 | 0.2978 | 0.5920 |
| KDD99 | 0.7898 | 0.2636 | 0.7882 |

method outperforms standard, optimally-configured RepStream, which as we mentioned before is likely due to the dynamic nature of our method, and the ability to use different $K$ values over time.

## 4.4 Discussion

As we noted in Section 4.1 a major problem with stream clustering algorithms is the sensitivity to user-set initial parameters. Our dynamic $K$ adjustment method allows for the internal $K$ value of RepStream to change over time in response to changes in the data distribution.

Table 4.3 shows the $F$-measure value RepStream configured at the optimal $K$ value, in terms of $F$-measure score, the worst possible $K$ value, as well as our dynamic $K$ method. Our method is configured using the same initial $K$ parameter used in the worst $F$-measure column of the table. The exact $K$ values used can be found in Table 4.2. As can be seen, however, our dynamic $K$ method allows significant improvements over RepStream configured at the same values.

Our method works in improving the results and making the RepStream algorithm less sensitive to initial parameters. Through all of our experiments we've used the same parameters - $\alpha$ scaling factor is set to $\alpha = 1.5$, vanilla normalisation, decay factor at the default $\lambda = 0.99$, and an initial $K$ set to the worst possible initial $K$ value for each dataset. For almost all instances this initial $K$ value was set to 5, as per Table 4.2, and so we would suggest that using an initial $K$ of 5 and letting the value automatically stabilise over time would be a satisfactory way to configure our method. As such our algorithm can perform without need for further tuning as it will automatically adjust its own internal $K$ parameter according to the computed average edge distribution score.

## 4.5 Conclusion

In this chapter we have introduced a method for automatic parameter selection in RepStream using edge distribution as a computed measure. RepStream (Lühr & Lazarescu, 2009) is a sophisticated clustering algorithm, employing a combination density and graph-based clustering approach, but one notable problem with it, and other stream clustering algorithms, is the reliance on user-set parameters. Here, we have extended the RepStream algorithm, proposing changes which remove the need for users to tune parameters to the dataset. Since data clustering is an exploratory processes this is particularly important, because one can't assume prior knowledge about the data to be analysed.

Our method consisted of making use of the $K$-nearest neighbour directed sparse graph employed by the RepStream algorithm and computing the edge distribution score to determine how well the distribution of edge lengths matches the expected distribution. With this measure we gradually raise or lower the $K$ value over time to keep the distribution score close to the appropriate level, represented by a threshold, which is set to minimise the number of edges that span between classes whilst maximising connectivity.

Our edge distribution score, described in detail in Section 4.2, is a computed measure which reflects how closely the edge lengths resemble what we would expect from a stable cluster, and makes use of the fact that areas of relatively continuous density have much less variation in the length of edges in a $K$-nearest neighbour context. Using this measure we propose a method for increasing and decreasing the $K$ value over time to adjust to changes in the distribution of data points in a stream.

Our experiments in Section 4.3 showed that when our method was configured using the $K$ value which would perform poorest, in terms of $F$-measure, in standard RepStream, our dynamic $K$ method was able to recover and produce significantly improved clustering results, shown in Table 4.3. We propose that using our method, and an arbitrary low initial $K$ value of $K = 5$ we can produce clustering output which is of consistent quality and which matches or even outperforms other sophisticated stream clustering algorithms, with respect to purity, when those algorithms are run with recommended parameters.

This method replaces the need for the user to configure the sensitive $K$ parameter and replaces it with a threshold parameter which is more robust to changes in distribution, and less sensitive to different datasets. In the next chapter we build on the ideas presented here, and integrate them more tightly into the algorithm.

# Chapter 5

# RobustRepStream - Graph-based Stream Clustering with Local Adaptivity

## 5.1 Overview of RobustRepStream

In the previous chapter we introduced a method for the $K$ parameter of RepStream to be automatically varied over time in order to adapt to changes in the distribution of the data stream to be clustered. This increased the robustness of the algorithm, reducing the need for user input while retaining high quality clustering output. In this chapter we seek to build on this idea, further increasing the robustness of the algorithm by removing the need to set the critical $K$ parameter altogether.

Data stream clustering, for example the STREAM (Guha et al., 2000) and CluStream (Aggarwal et al., 2003) algorithms, are some of the earliest examples, have unique and hard-to-address challenges as compared to clustering static batch data sets. In chapter 1 we describe how the aim of clustering is to take a data stream represented by a series of $d$-dimensional data points $X_1, X_2, ..., X_i...$ and to map each data point to a cluster $C_1, C_2, \ldots C_j \ldots$, with the aim of data points which are similar according to some distance metric being grouped into the same cluster.

A specific and major challenge in data clustering is setting appropriate input parameters. Many algorithms require the user to set initial hyper-parameters which affect the performance of the algorithm. Poorly selected values for these parameters can lead to poor clustering performance, with the clustering algorithms not discriminating between groups in the data well enough (over-clustering) or fracturing groups of data into smaller clusters than is desirable (under-clustering).

Having a robust algorithm, i.e. being able to perform well with unpredictable changes in data distribution, is critical in stream clustering since data distributions may change significantly over time. For example, data samples may shift their typical values, data density becomes denser or sparser, clusters may split into smaller clusters or join with other clusters, etc. As such, an algorithm that relies on data-dependent input parameters usually faces the problem of the selected values being inappropriate later during a stream. Even if a user hypothetically could optimally parametrise an algorithm at the beginning, it could still produce poor quality clustering at later points in the stream, due to concept drift (Widmer & Kubat, 1996).

It is therefore desirable to have a robust clustering method which is less sensitive to input parameters and perform well on an evolving data stream, or even to reduce the number of parameters which must be set by users, where possible. Unfortunately, most algorithms typically require careful selection of the input parameters based on knowledge of the data set and do not have such a built-in mechanism to vary these parameters at run time when significant changes in the data stream occur. For example, the *k*-means based clustering approach (Zhou et al., 2008) requires the exact number of clusters *k*, grid-based approaches (Wan et al., 2009; Chen & Tu, 2007; Lee, 2016) require a density threshold for cells and grid granularity values, and micro-cluster density-based approaches (Cao et al., 2006; Hahsler & Bolaos, 2016) require either density thresholds or micro-cluster radii to be user specified. When optimal selection of these parameters cannot be made due to lack of knowledge about the data distributions, which is typically the case with stream clustering, these methods often perform poorly.

The original version of RepStream (Lühr & Lazarescu, 2009) is sensitive to the *K* parameter which determines the level of connectivity. Once specified at the beginning, the *K* parameter in the original RepStream algorithm remains fixed through the entire runtime of the algorithm and therefore it does not adapt to changes which may occur later. The extension to RepStream which we proposed in the previous chapter adapted the *K* parameter automatically, which allowed the algorithm to adapt to changes in the data distribution.

In this chapter we present the RobustRepStream algorithm, which allows for dynamic levels of connectivity for each vertex in the point and representative level sparse graphs rather than relying on a single, set $K$ parameter. The usage of a universal $K$ value carried the assumption that a single level of connectivity was appropriate at all regions in the data space, which may not be a safe thing to assume when there is any variation in distribution properties between ground truth classes. Instead, in RobustRepStream each vertex has its own connectivity level to neighbours determined by the distribution of neighbours in its local area. This provides the benefit of no longer needing to set the $K$ parameter at all, and also means that each vertex can have different levels of connectivity where appropriate.

We first introduce novel skewness excess scores, describing the abnormality level in the distribution of edge lengths at each vertex. Our intuition is that optimal clustering is the right balance between the graph complexity for this class of algorithms, which is described by its connectivity, and the skewness excess: one should aim at a high graph connectivity value whilst maintaining the edge distribution skewness excess under an acceptable level, which is universal across data streams. We achieve this dynamic connectivity by finding a maximum connectivity value at which the the graph skewness excess is still within a pre-defined universal abnormality threshold. This method improves on our previous algorithm by being more robust at lower connectivity levels by computing edge variance using median absolute deviation, and also by simplifying the input parameters, removing the separate $\theta$ parameter in DRepStream. By doing so, our new algorithm can determine the suitable level of connectivity for each vertex at any point in time. Experimentally, we demonstrate over benchmark data sets for data stream clustering that RobustRepStream is robust as it produces more consistent clustering results using the same input parameters between different data sets. This chapter contains the following contributions:

- A novel concept, termed *skewness excess*, which is useful for adapting $K$-NN graphs to changes in data stream statistics, together with theoretical insights and a method based on the median absolute deviation (MAD) to compute them which is robust at smaller connectivity values.

- Our RobustRepStream algorithm which automatically adapts the internal graph to the changes in data streams statistics by balancing its graph connectivity and graph skewness excess. This results in a nearest-neighbour graph whose connectivity is optimised locally to adapt to the changes in the data streams. Consequently, our proposed method removes the need for the user to specify the connectivity parameter $K$.

- Comprehensive evaluations against recent stream clustering algorithms using benchmark data sets specifically for data streams.

This chapter is organised as follows. Section 5.2 introduces a new concept, the skewness excess, which is a feature that can be extracted from directed sparse graphs for each vertex in the graph, and we also describe our proposed RobustRepStream algorithm, which extends RepStream by using the skewness excess to dynamically select outgoing neighbours in its directed sparse graph structures. Our experiments evaluating RobustRepStream are detailed in Section 5.3 against well-known stream clustering algorithms. Our results are discussed in more detail in Section 5.4, and finally we present our concluding remarks in Section 5.5.

## 5.2 Proposed Method

As can be seen in chapter 2, the performance of RepStream depends largely on how well the connectivity parameter $K$ is specified by the input. The value of this input parameter is significantly influenced by the statistics of the underlying data: it will over-cluster data points if $K$ is too small, and under-cluster data points if $K$ is too large. Unfortunately, RepStream does not have the ability to tell which $K$ value is appropriate and hence relies totally on the user's input, much like many other $K$-NN-based clustering algorithms. In chapter 4 we demonstrated a way to automatically vary $K$, in this chapter we aim to remove the need to set a universal $K$ connectivity parameter at all.

To address the limitation of RepStream, we study the distribution of edge lengths over the $K$-NN graphs and how it depends on the connectivity parameter $K$. We follow the fundamental principle in machine learning, which is to select the simplest model that explains the data well. Here, the model complexity is inversely proportional to $K$ as described above, and the explain-ability of the model is, according to our definition, how well the distribution of the normalised edge lengths still looks 'normal'. By searching for the simplest model that still controls the deviation of the normalised edge length distribution under a pre-defined level, we can automatically adjust clustering outputs when the underlying data statistics changes. Based on the RepStream algorithm, our proposed RobustRepStream consists of two main parts:

- The computation of the skewness excess score of the distribution of normalised edge lengths at each vertex.

- Automatic model selection based on the computed skewness excess score.

### 5.2.1 Skewness Excess

To motivate our approach, we start with an intuition shown in Figure 5.2. Typically, in a volume of even distribution of data points the edges will be more likely to be spread evenly in all directions. A data point in the middle of a class has other vertices belonging to that class all around it, and so the variation in edge length of its neighbours will be relatively small. In the middle of a cluster there are data points in all directions, so there are likely to be more available neighbours at relatively even distances. In this example, the vertex $R_2$ is near the centre of the class $C_2$ and so many intra-class edges can be made, even if the connectivity of $R_2$ is increased. On the other hand the vertex $R_1$ is near the edge of the class, and so the potential intra-class neighbours is fewer. There are limited directions in which nearby intra-class edges can be made, and the other directions border the areas of low density which, as we discussed before, separates clusters. Any inter-class edge must traverse this distance, which means the edge length will likely be longer than inter-class edges. Vertices on the edge of a cluster are therefore more likely to have longer, less consistent edge lengths for neighbours.

Suppose that desirable clusters are regions of contiguous (but not necessarily convex) space which are populated by data points in a more or less similar distribution. These regions we assume are separated either by significant regions of empty space, or by data points at a significantly lower density than inside the clusters. From this assumption, it follows that data points near the edges (but still inside) a cluster are very likely to have the first few of their nearest neighbours belonging to the same cluster, and that the chance of the next nearest neighbour belonging to a different class (an inter-class neighbour) increases as the number of nearest neighbours being considered increases. This means the connectivity needs to be adjusted to reflect the desirable clustering outcome.

As can be seen, once all the nearby vertices within a multi-dimensional sphere have been connected to, additional neighbours will have a greater and greater likelihood of being significantly further away than existing neighbours. It is these longer than average edges which the score is intended to reflect, and high numbers of large edges suggest that the number of edges has grown too large. They are more likely to be on the long tail of the distribution of edge lengths. Note that as the edge lengths are non-negative, the more extreme values of edge lengths associated with a vertex, the more skew to the right the distribution of the edge lengths. Therefore, the excess at the right

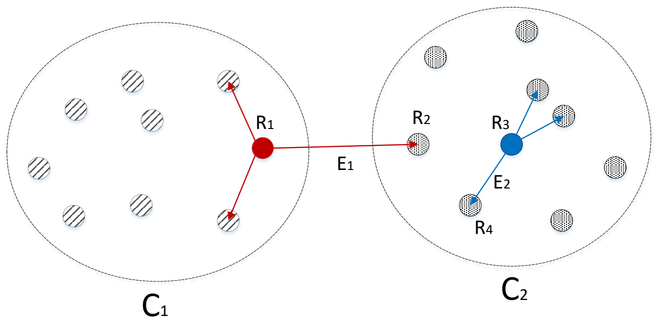


Figure 5.1: Intra and Inter-class edges. The Edge $E_1$ is considered an inter-class edge as it connects two vertices $R_1$ and $R_2$ that belong to different ground-truth classes. Edge $E_2$ connects two vertices belonging to the same class and thus is considered an intra-class edge.

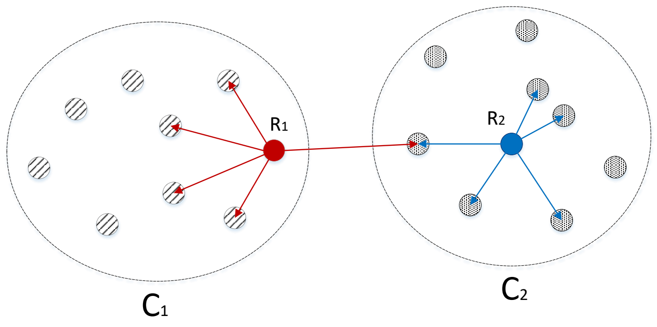


Figure 5.2: An illustration of relative edge lengths of nearest neighbours in the middle of a cluster versus near the edge of a cluster.

tail of the distribution of the edge lengths captures the presence of these large values. Based on this intuition, we introduce a novel concept of skewness excess.

In what follows, we denote as $v_i$ the vertex $i$ of interest, $e_{i,j}$ as an outgoing edge from vertex $v_i$ to another vertex in its neighbourhood $\mathscr{N}_{K_i}(v_i)$. We also denote as $X_{e_{i,j}}$ the *normalised* length of the edge $e_{i,j}$. Here, normalisation is performed with respect to the distribution of original outgoing edges from vertex $v_j$

$$X_{e_{i,j}} = \frac{l_{e_{i,j}}}{1.4826\nu_i}, \tag{5.1}$$

where $l_{e_{i,j}}$ is the original length of the edge $e_{i,j}$ and $\nu_j$ is the median of the absolute deviation (MAD) of the $K$ values of $l_{e_{i,j}}$. The reason we use $1.4826\nu_i$ is because it is a robust measure of the deviation of the edge lengths (Rousseeuw & Croux, 1993). Given that a vertex may have a relatively small number of outgoing edges - fewer than 10 - the standard deviation is not a reliable measure.

To capture the tail behaviour of the edge length distribution, we define the thresholding function

**Definition 5.2.1** *The thresholding function $\rho_\tau(x)$ with a threshold $\tau$ is defined as*

$$\rho_\tau = \begin{cases} x & if\, x > \tau, \\ 0 & if\, x \leq \tau. \end{cases} \tag{5.2}$$

Denote as $\bar{X}_{v_i}$ the median of all normalised edge lengths associated with $v_i$. We formally introduce the following concept

**Definition 5.2.2** *The skewness excess score (SE) of $K_i$ normalised edge lengths associated with a vertex $v_i$ is*

$$SE(v_i) = \sum_{e_{ij} \in \mathscr{N}_{K_i}(v_i)} \rho_1(X_{e_{i,j}} - \bar{X}_{v_i}). \tag{5.3}$$

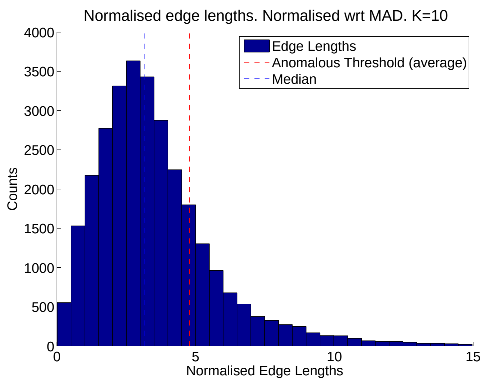
Note that the subscript $i$ in $K_i$ indicates that $K_i$ is locally specific to each vertex, as opposed to a single global $K$ in RepStream and most $K$-NN data stream clustering algorithms. Here, the skewness comes from the older notion of non-parametric skew, and the excess is due to our choice of considering only edges with normalised length greater than the median by 1. Edges that exceed this length will be considered in the overall skewness score. Consequently, additional neighbours must be within a relatively limited distance of existing neighbours, otherwise the edge will be considered anomalous. As the number of outgoing edges, and neighbours therefore get farther and farther away, the total skewness excess tends to increase as shown in Fig 5.4.

Given that the area of a hyper-sphere increases exponentially as the radius is increased, the variance in edge length of additional neighbours should decrease. Figure 5.6 shows an example of the 200 nearest neighbours of a vertex in a 2 dimensional normal random distribution consisting of 400 vertices in total in which the standard deviation is 100. Figure 5.5 shows a histogram of the length of the edges to these nearest neighbours.

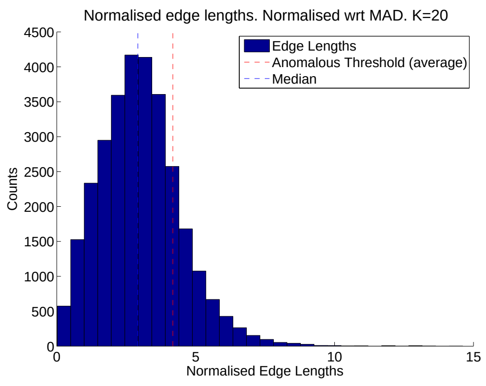
This example follows our expected trend, in which the distribution of edge length is weighted towards the far end. With this knowledge we expect a vertex in the middle of a cluster will have a more concentrated distribution, whereas a vertex near the edge of a cluster will have a more right-tailed distribution. This is the key principle of our proposed RobustRepStream.

Next, we briefly study the asymptotic behaviour of the skewness excess score through a real example when both $K$ and the number of vertices are sufficiently large. We sample 2-dimensional data from a uniformly random distribution, and define a point directly in the centre of this distribution as our evaluation point, and run our proposed method. At a point in the middle of the stream, we examine the average of the distributions of the normalised edge lengths for two different values of $K$(Figs. 5.3(a) to 5.3(c) ) as well as the average skewness excess score calculated over *all* vertices (Fig. 5.4). Here, we make two interesting observations. The first observation is that the average distribution exhibits a linear behaviour for edge length below the median, and exponential decay beyond the median for both $K = 20$ and $K = 50$. The tail behaviour is also consistent with our intuition: when $K = 10$ (small), there are relatively more large values at the tail as the model is more 'complex' and tends to produce smaller clusters with less data points. Consequently, a large edge length will be more likely to be anomalous compared to the rest. When $K = 50$, it is less likely for an edge to be relatively large compared to the rest, and therefore the distribution is more concentrated around the median.

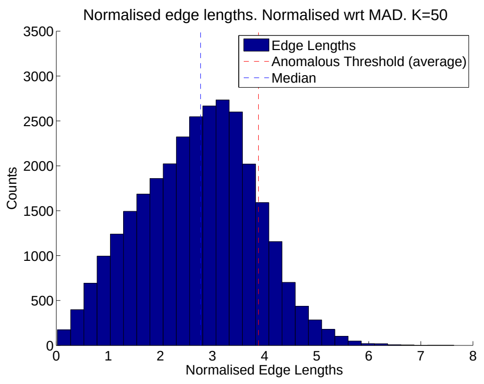
The second observation is that the average skewness excess score exhibits an approximately linear relationship with respect to $K$ over the practically meaningful range of values for $K$. This is also consistent with our intuition, because on average the term in the sum converges to the first-order moment of the distribution of the normalised edge lengths partially over the support beyond $1+\nu_i$, and the fact that there are $K$ terms in the sum. This linear asymptotic behaviour is important because it allows us to pro-



(a) Average histogram of normalised edge lengths for $K = 10$



(b) Average histogram of normalised edge lengths for $K = 20$



(c) Average histogram of normalised edge lengths for $K = 50$

Figure 5.3: Histograms of normalised edge lengths for various $K$ values

vide a trade-off between model complexity and data explain-ability more easily: the larger the $K$, the less complex the model (less clusters), but the more discrepancy with data statistics (more inter-class pairs in a cluster). Due to the linear relationship, a one-dimensional search for the optimal value of $K$ will be feasible without a risk of facing multiple local optimum values, at least in the asymptotic sense. This is what we propose do exploit next.

### 5.2.2 RobustRepStream

The RobustRepStream algorithm is an extension of the RepStream algorithm detailed in the previous section. RobustRepStream, uses a completely distinct method for selecting outgoing edges for each vertex, at both the point and representative levels. The skewness excess score as described previously is used to determine how many edges

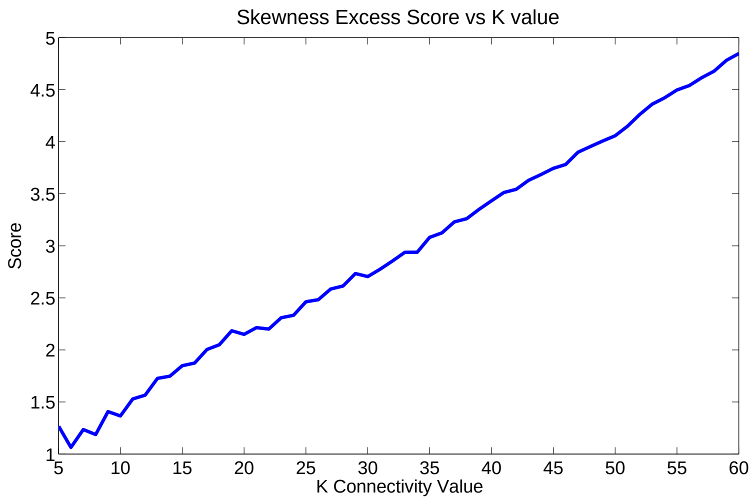


Figure 5.4: Average SE vs $K$, normalised with respect to MAD

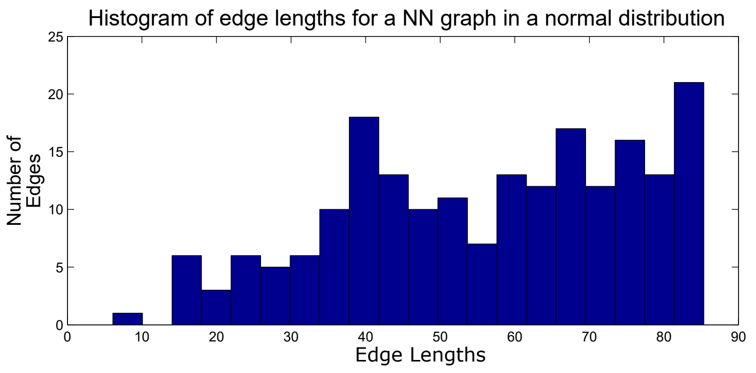


Figure 5.5: Histogram showing the distance to the nearest 200 vertices in a 400 point 2 dimensional normal distribution, with the origin at the centre of the distribution. Standard deviation is 100 in both the x and y direction.

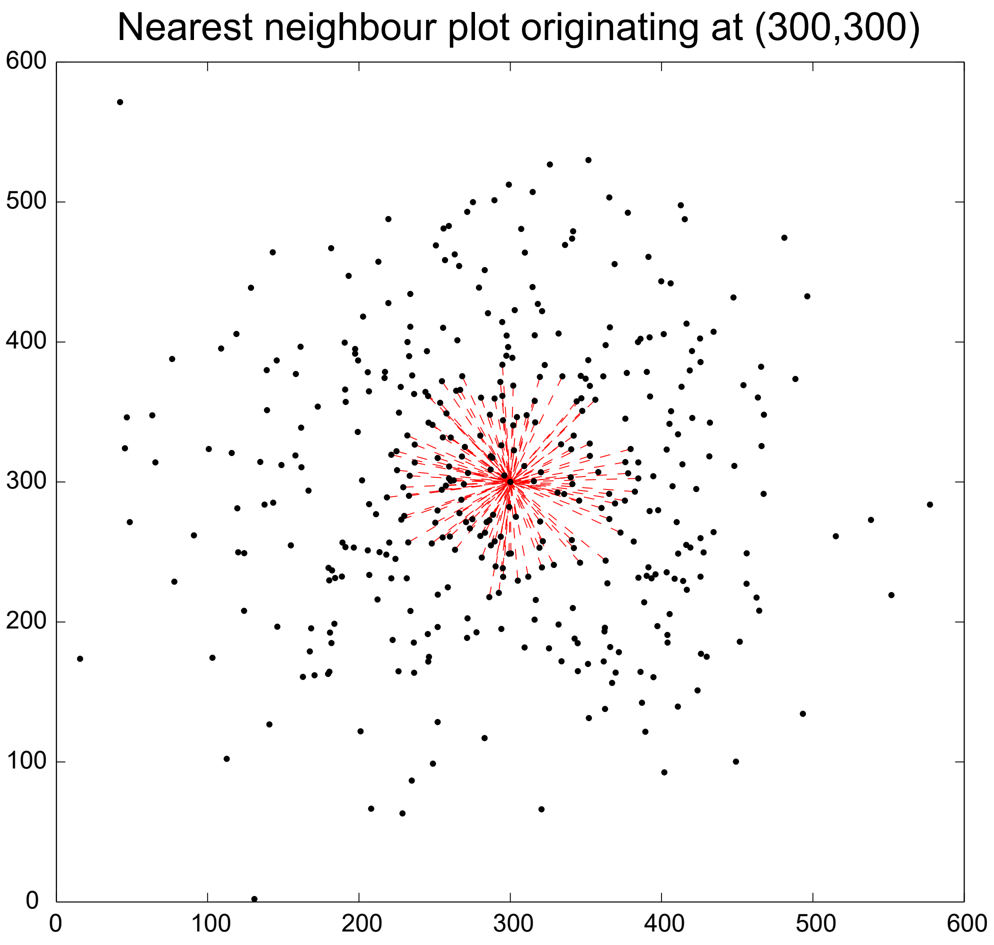


Figure 5.6: Nearest neighbour plot of a 400 point 2 dimensional normal distribution with a standard deviation of 100. Plot shows the 200 nearest neighbours from the centre of the distribution.

each vertex should have when it is inserted into the graph. In the original RepStream algorithm, every vertex will always have the *same* number of edges connecting to other vertices, regardless of the relative distance to the farther neighbours compared to closer neighbours. Not only must a user correctly select a single $K$ value for use during an entire stream's length, but the inflexibility in neighbour counts means one must make the trade-off between over-clustering and under-clustering. To address this limitation, we aim to achieve two goals with the proposed algorithm:

*Goal 1: Remove the need for the user to set a the sensitive K parameter at runtime.* As we have mentioned in previous sections, initial parameter selection is a challenge, and our method aims to converge towards a desirable clustering solution.

*Goal 2: Allow vertices in the two NN graphs to have dynamic and different levels of connectivity as is appropriate for their local region.* Setting outgoing edges in a dynamic manner allows the construction of a nearest neighbour graph which takes the context and distances of the nearest neighbours into account. Allowing vertices in the nearest neighbour graph to have different numbers of outgoing edges will give us the potential to allow different levels of connectivity in denser regions of data, compared to uneven or sparse regions.

The RobustRepStream works on the principle of balancing between model complexity and data explain-ability and it proceeds as follows. For any newly inserted vertex $v_i$ we aim to find an optimal value of the $K_i$ connectivity parameter locally specific to the vertex. To so do, we do a one-dimensional search, which starts from a pre-defined value $K_i = K_i^{\min}$ and gradually increases it. At every possible value of $K_i$, we examine the skewness excess score of the distribution of the $K_i$ normalised edge lengths associated with $v_i$. The selected $K_i$ is then the maximum value at which the skewness excess score is still below a universal threshold $\alpha$ describing the maximum discrepancy between what the model assumes and the actual data

$$K_i = \arg \max_{K > K_i^{\min}} SE(v_i) \leq \alpha. \tag{5.4}$$

The result of this is that each vertex will have the maximum number of neighbours possible, whilst still remaining under the skewness excess score threshold. When a vertex is added or removed, nearby vertices must also be adjusted by either adding or removing outgoing edges until the number of edges is the highest possible while remaining below the maximum allowable threshold.

Compared to the original RepStream algorithm and many other $K$-NN-based clustering methods, RobustRepStream has removed the global connectivity parameter $K$. Whilst our proposed algorithm has introduced the new threshold $\alpha$, it is much less sensitive to the data unlike the connectivity parameter, which is clearly demonstrated in the sensitivity analysis we show later. Because it is based on the distribution of the normalised edge lengths, it is invariant to the scale of the data. It is purely a subjective definition of what an acceptable level of excess skewness by the user. In fact, a universal value $\alpha = 2$ is recommended for all data sets. The proposed RobustRepStream method consists of two algorithms:

- The first algorithm determines the number of outgoing edges for a newly added vertex as shown in Algorithm 5. In this algorithm, the inputs are $vertex$, the new vertex, $minEdges$ the minimum number of edges a vertex must have, and $\alpha$ is the density existing scaling factor parameter used by RepStream. The function $GetNextNeighbour(vertex)$ gets the closest neighbour that the vertex does not already have an edge to, and the function $CreateEdge(vertex, newNeighbour)$ creates an outgoing edge from $vertex$ to $newNeighbour$.

- The second algorithm calculates the skewness excess score as if the vertex $newNeighbour$ were to hypothetically become a neighbour, and is shown in Algorithm 6. In this algorithm, the average and median absolute deviation values are calculate as $avg$ and $dev$ respectively with the potential new edge, then the edge anomaly score for the potential edge is computed and returned. The function $SumDistances(vertex.neighbours)$ finds the sum of the distances to all existing neighbours, while the function $Distance(vertex, neighbour)$ finds the distance between two vertices.

Once a suitable number of outgoing edges is found so that the skewness excess is below the threshold $\alpha$, RobustRepStream assigns clusters using the same mechanism as the original RepStream algorithm.

The existing $\alpha$ parameter is used due to its existing usage in RepStream for determining density relation, as described in the previous section. The density relation radius is equal to the average length of the vertex's nearest neighbours multiplied by the $\alpha$ parameter, and defines an upper limit to the distance between representative points for them to be merged into the same cluster. Our skewness excess threshold follows a similar concept, being an upper-limit to the relative length of edges to nearest neighbours.

**Algorithm 5** Algorithm for determining the outgoing edges for a newly added vertex.

f

```
$FUNCTION : AddEdges$
$INPUT : vertex, minEdges, \alpha$
**while** $vertex.neighbours.count < minEdges$ **do**
    $newNeighbour \leftarrow GetNextNeighbour(vertex)$
    $CreateEdge(vertex, newNeighbour)$
**end while**
$newNeighbour \leftarrow getNextNeighbour$
**while** $SkewnessExcessScore(vertex, newNeighbour) < \alpha$ **do**
    $CreateEdge(vertex, newNeighbour)$
    $newNeighbour \leftarrow getNextNeighbour$
**end while**
```

**Algorithm 6** Algorithm for calculating the skewness excess score of a vertex if a potential new neighbour were to be added.

```
$FUNCTION : SkewnessExcessScore$
$INPUT : vertex, newNeighbour$
$score \leftarrow 0$
$dev \leftarrow 0$
$PotentialNeighbours \leftarrow \{\}$
$PotentialNeighbours \leftarrow vertex.neighbours + newNeighbour$
$avg \leftarrow Median(PotentialNeighbours.distances)$
$AbsDevs \leftarrow \{\}$
**for all** $neighbour$ in $PotentialNeighbours$ **do**
    $AbsDevs.add(|Distance(vertex, neighbour) - avg|)$
**end for**
$dev \leftarrow 1.4826 \times Median(AbsDevs)$
**for all** $neighbour$ in $vertex.neighbours$ **do**
    $s \leftarrow 0$
    **if** $Distance(vertex, neighbour) - avg > dev$ **then**
        $s \leftarrow (Distance(vertex, neighbour) - avg) \div dev$
    **end if**
    $score \leftarrow score + s$
**end for**
**if** $Distance(vertex, newNeighbour) - avg > dev$ **then**
    $s \leftarrow (Distance(vertex, newNeighbour) - avg) \div dev$
    $score \leftarrow score + s$
**end if**
**return** $score$
```

As is similar to its original usage in RepStream, $\alpha$ is a relatively insensitive parameter which is much easier to set than the $K$ value. Intuitively, an alpha value of $\alpha \leq 1$ makes little sense since representative vertices are only created when a vertex has no reciprocal link to an existing representative, and so representative vertices are most likely to be spread out. Skewness excess threshold values less than 1 also make little sense because the edge score cannot have a value between 0 and 1.

We found empirically that the universal choice $\alpha = 2$ works well across many types of data streams. We also define a lower bound to the number of edges as being $minEdges = 6$, as fewer edges than this makes the computation of median absolute deviation less meaningful. This is what we use in our experiments in Section 5.3, regardless of data set. An $\alpha$ value of 2 allows vertices to have no more than 1 anomalous edge, as the minimum excess score of an anomalous edge is 1.

## 5.3 Experiments

To evaluate RobustRepStream we perform experiments on a number of real-world and synthetic data sets. These datasets are specifically selected to be examples of streaming data which evolves over time, exhibiting concept drift with which we can evaluate our RobustRepStream method.

### 5.3.1 Real World Data Sets: KDD and Tree Cover

**The KDD Cup 1999 data set** The KDD'99 data set (Hettich & Bay, 1999) is a well-known benchmark data set. It is extracted from logs taken from a smart firewall in a network being subjected to simulated and controlled network attacks. It contains high dimensional data, of which we use the 34 numerical features with each data point presented as a 34-dimensional vector. We use a subsampled version of the data set containing 494 020 data points, which is about 10 % of the original KDD Cup 1999 data set. Most of the data in the subsampled data set we used for evaluation falls into either the normal traffic class, or one of two major denial-of-service attack classes. A relatively small percentage of the data - less than 2 % - are from 20 other network attack types. Each data point is labelled with the type of traffic (normal, or the type of attack) for evaluation purposes.

This KDD Cup 1999 data set has been used previously in evaluating stream clustering algorithms (Ackermann et al., 2012; Aggarwal et al., 2004; Bhatnagar et al., 2014; Lühr & Lazarescu, 2009) due to the high variability between classes in the data set. The various network attacks interrupting the normal traffic represent changes in the distribution of subsequent data points, known as concept drift. This is a significant challenge for clustering algorithms to deal with, making it an excellent data set for testing how an algorithm deals with dynamic, unpredictable data point distributions over time.

**The Tree Cover Type Data Set** The CoverType data set (Blackard & Dean, 1999) is a real-world data stream of a set of features extracted from satellite photos and geological surveys from forested areas of northern Colorado. It contains over 580,000 entries with ground truth labels corresponding to which type of trees grow in each area, and has been previously used as a benchmark data set for stream clustering (Lühr & Lazarescu, 2009; Bhatnagar et al., 2014; Forestiero et al., 2013). This data represents a naturally evolving stream of data which changes with the environment and climate of each region. As such, this data set, as well as the previous KDD data set, both contain real-world examples of concept drift in a streaming context. The dataset has 10 dimensions which include quantitative measures of elevation, aspect, slope, distance from water source, distance from roadway, and level of incident sunlight. Whilst this stream evolves over physical location rather than over time, it is still effective for use as a benchmark dataset exhibiting concept drift. Both this data set and the KDD dataset contain a level of natural noise due to their respective collection methods.

### 5.3.2 Synthetic Data Sets

The synthetic data sets we present here are designed to evaluate our algorithm on controlled levels of concept drift, which present specific challenges to the algorithm and demonstrate its ability to handle change over time. As such, these data sets include data sets which require variable levels of connectivity in order to achieve the highest quality clustering results. Our intuition is that graph-based clustering approaches perform better using a lower level of connectivity (fewer outgoing edges) when there is less separation between ground truth data classes, and more connectivity (more outgoing edges) when there is a greater distance between data classes. This is to prevent over-clustering at levels of low separation, and to prevent under-clustering when there is high separation between classes.

**Closer Data Set** The Closer data set is an evolving data set which has three distinct stages. The first 10,000 data points alternate between two classes, all data points are

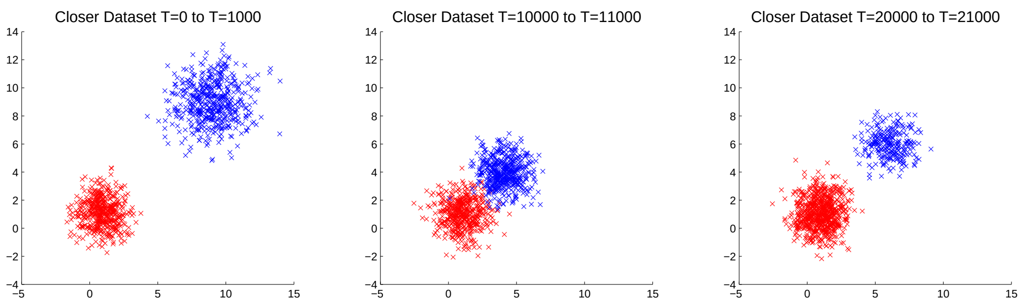


Figure 5.7: The evolution of the Closer data set, showing slices of its 3 sections.

on a two-dimensional plane and each class is normally distributed with a large level of separation between the two classes. In the second 10,000 data points, the two classes suddenly become much closer together, such that the two classes borders are overlapping. The final 10,000 points have a greater degree of separation once again, however one class becomes more dense, while the other becomes less dense. Figure 5.7 shows these three stages. The changes in this data set are sudden, being taken from only three different distributions.

The three stages of the data set were sampled like so:

- Between $T = 0$ to $T = 10,000$ class A was centred at 1,1 and normally distributed with a $\sigma$ of 1 in both the x and y axes. Class B was centred at 8,8 with a $\sigma$ of 1.5 on both axes. Points were sampled from these distributions alternately between classes.

- Between $T = 10,000$ to $T = 20,000$ class A remained the same, while class B was moved to 4,4. Points were sampled from these distributions alternately between classes.

- Between $T = 20,000$ to $T = 30,000$ class A remained the same, while class B was moved to 6,6 with a $\sigma$ of 1.5 on both axes. Points were sampled from these distributions randomly by alternately selecting 3 points from class A, followed by 1 point from class B.

**SynTest Data Set** The SynTest data set is also an evolving data set that consists of one large class which slowly shifts its shape and position over time, as well as several other smaller, more dense classes which appear and disappear at various points. The larger class is present throughout the whole data set, and makes up a majority of the

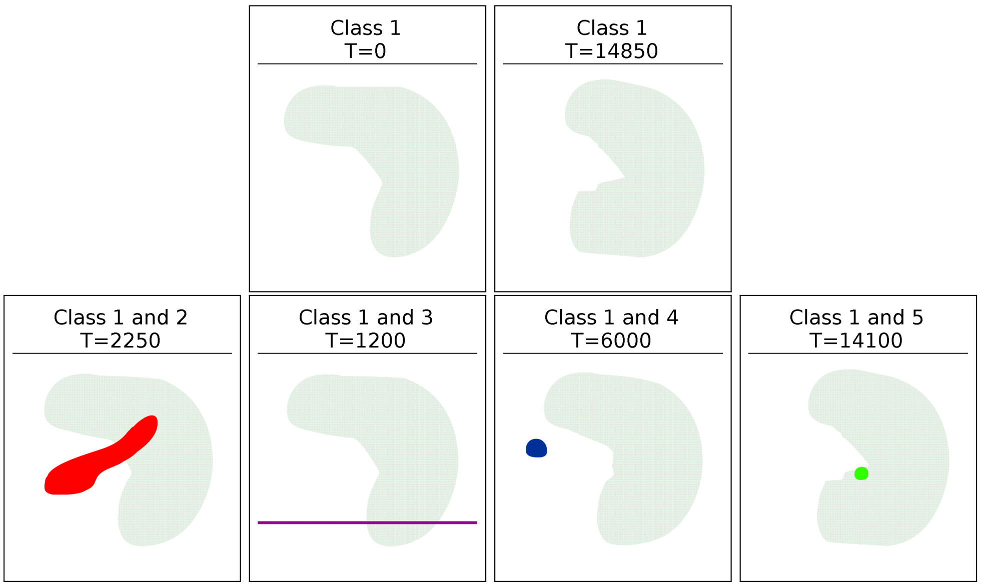


Figure 5.8: Two dimensional representations of the 5 different classes. The main class is always present and steadily changes shape, the smaller classes appear at various points through the data set, as shown in Figure 5.9.

data points, the smaller classes exist for a relatively shorter amount of time. Each of these smaller classes are more dense than the main class, but are present for only a few hundred, to a few thousand time-steps at a time.

Figure 5.9 shows the presence of the classes in the SynTest data set. Marks indicate when the given class is present in the given time window. The shape, size, and position of the classes is shown in figure 5.8. Class 1 is always present through the data set, while the other classes are present for shorter time periods.

**Shapes Data Set** The Shapes data set is an evolving data set split into two distinct stages made from two distributions. In the first stage, shown in Figure 5.10, the data set shared amongst 6 classes which are largely separated. In the second stage, the classes are moved closer together, and the two classes on the right of the data set merge to become one larger data set. The transition between these two stages presents a significant challenge to clustering algorithms, as a lower degree of separation between classes makes those classes harder to distinguish. Additionally, the two classes that are merged together means that the algorithm will have to adapt and merge clusters that were previously separated.

**DS1 and DS2** The DS1 and DS2 data sets are data sets made from a static distribution which were used to evaluate the original RepStream algorithm. They are shown

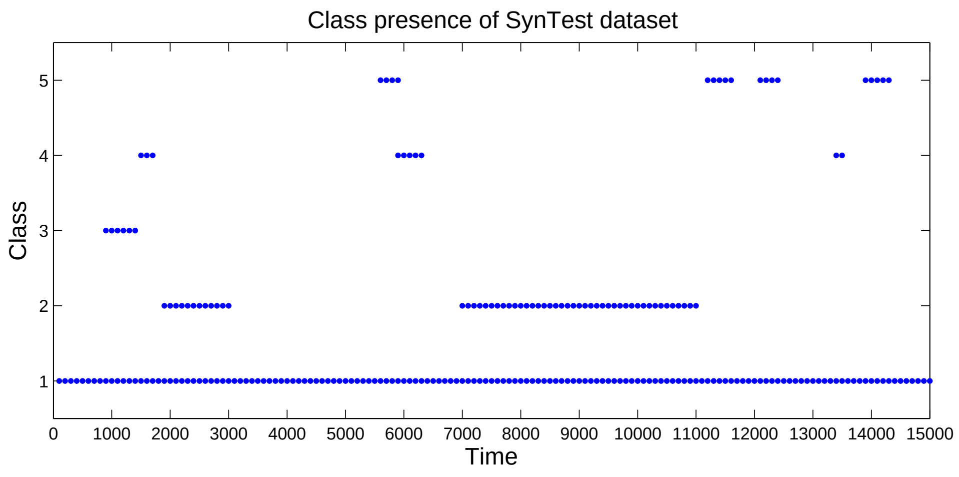


Figure 5.9: The class presence of the classes in the SynTest data set. A marker indicates the class is present in the data set during the given time window.

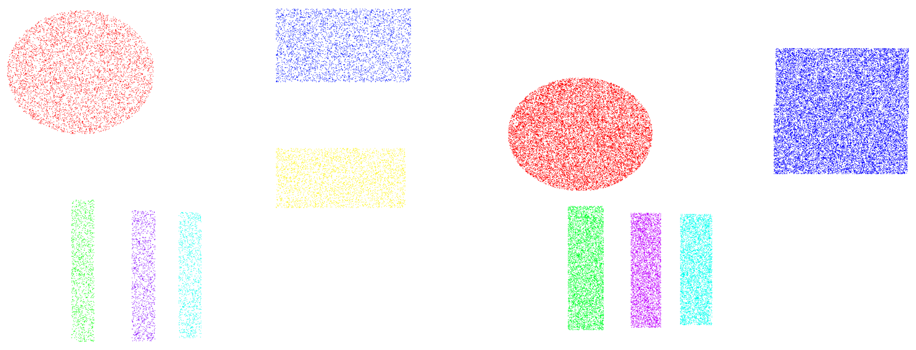
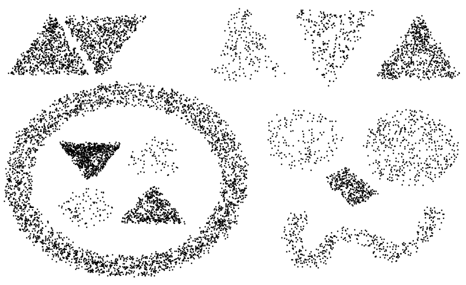


Figure 5.10: The first and second stage of the Shapes data set.


(a) Visualisation of the DS1 data set.

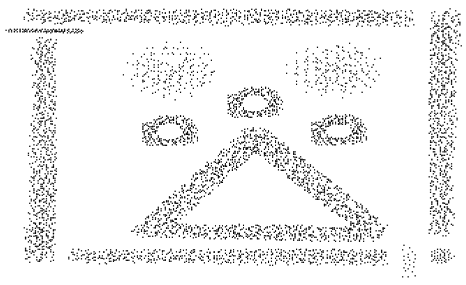

(b) Visualisation of the DS2 data set.

Figure 5.11: DS1 and DS2 datasets.

in Figures 5.11(a) and 5.11(b) and have no intentional stream evolution over time.

The DS1 and DS2 data sets are included as static data sets to demonstrate our method's ability to adapt to a static distribution, and to converge towards a stable clustering solution. The Closer, SynTest, and Shapes synthetic data sets all have specific and very controlled instances of concept drift over the length of the stream. These synthetic evolving data sets are used to demonstrate how our method can adapt to changing data distributions over time.

### 5.3.3 Evaluation Metrics

Since we have access to the ground truth class labels for the data set we are able to use external validation metrics as a way to determine the accuracy of our clustering methods. External validation metrics are more accurate in measuring the absolute quality of clustering results because they use a comparison to a ground truth. While internal validation metrics can be useful, they make assumptions about the data set which may be inaccurate, for example the sum of squared errors (SSQ) evaluates the compactness of clusters, with a lower score being more desirable. This measure assumes ideal clusters are hyper-spherical in shape, which is not a safe assumption, since clusters can be arbitrarily shaped.

We use the average $F$ measure score of clusters compared to the ground truth over the length of the data set as a method of measuring the performance of our RobustRepStream method against the base RepStream algorithm. We choose the external validation metric, $F$ measure since it can be used to compare clustering results, while penalising both different ground-truth classes being mixed into the same cluster, and also single classes being fractured into multiple clusters. The commonly used purity measure is more popular for evaluation, but has a problem in that it does not penalise classes being fractured into sub-clusters (Kaur et al., 2015). For this reason we find $F$ measure to be a more accurate representation of cluster accuracy than the more popular purity measure for evaluating RobustRepStream against the performance of RepStream in terms of both precision and recall. For comparisons against other algorithms we use the purity measure because of its ubiquitousness in the literature.

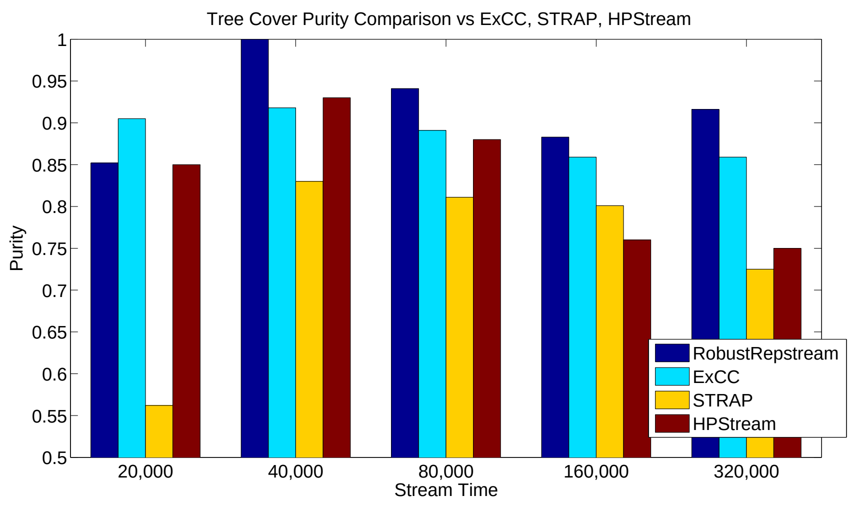
### 5.3.4 Comparison to Other Algorithms

To evaluate the comparative performance of our method we compare against the published results of other stream clustering algorithms. Namely, we compare against the base RepStream algorithm, as well as published results for ExCC (Bhatnagar et al., 2014), STRAP (Zhang et al., 2008), HPStream (Aggarwal et al., 2004), stream specific algorithms that have shown high quality performance in stream clustering context. The published results favour the purity evaluation metric , which is commonly used for external validation when ground truth class labels are available, as is the case with the KDD Cup and Tree Cover data sets. For our experiments we use a purity horizon of 200 data points, which is a common horizon for the evaluation of clustering algorithms when using purity (Aggarwal et al., 2003) (Aggarwal et al., 2004).

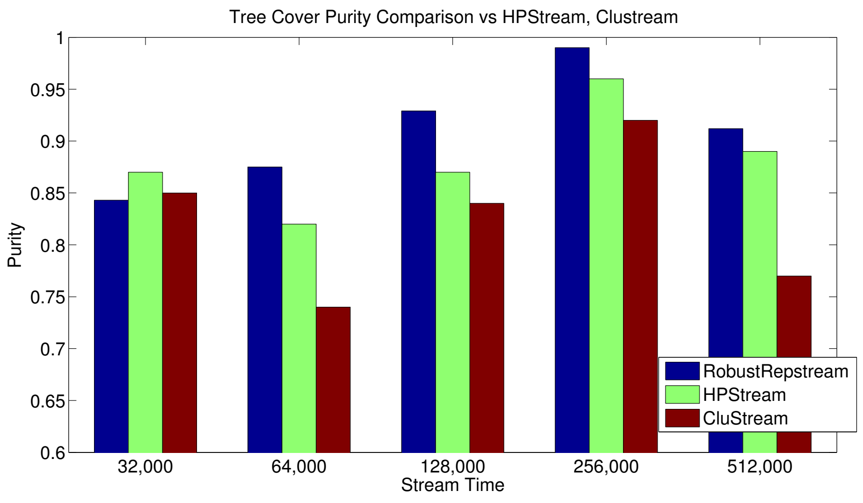
Figure 5.13(a) shows the comparative purity of the different methods on the KDD data set. RobustRepStream can be seen to perform favourably to the other clustering algorithms in all cases. HPStream results in equal purity at the 51 000 and 371 400 time slices, while STRAP outperforms RobustRepStream slightly during the 86 600 time slice. However, overall our method consistently produces high purity output during the KDD dataset, as we show in further evaluations.

Figure 5.12(a) shows the comparative purity for the Tree Cover data set. It is notable that RobustRepStream outperforms the other algorithms using the published results for these methods for all time steps save for the 20 000 time step. ExCC performs consistently highly on this dataset similar to RobustRepStream and has a lower variance in purity over time. However our method does overall outperform it on these published time steps. We show in subsequent evaluations that our method is able to maintain similar high levels of clustering output.

We also compare against the results of the D-Stream (Chen & Tu, 2007) and DB-Stream (Hahsler & Bolaos, 2016) algorithms, grid-based and micro-cluster density-based stream clustering algorithms respectively, which perform well against the well known CluStream and DenStream algorithms, as shown in Figure 5.13(b) and 5.12(b). D-Stream divides the data space into fixed-width cells, and tracks which cells become

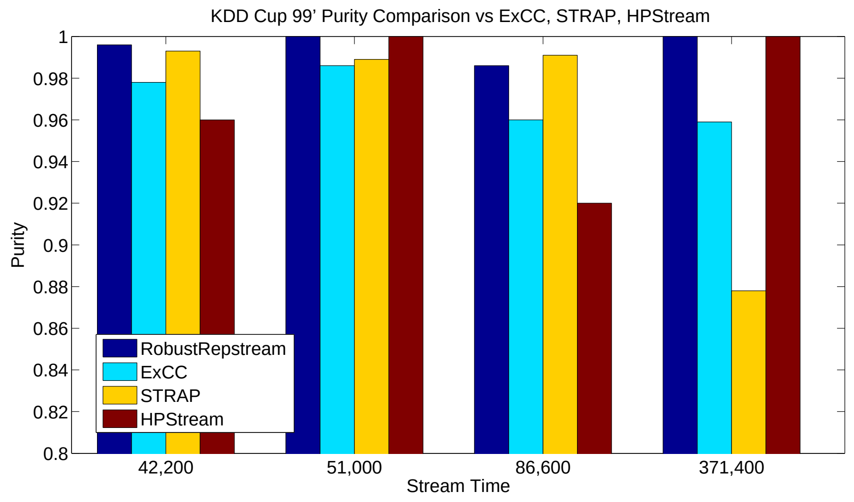


(a) Comparative purity for TreeCov data set against ExCC, STRAP, and HPStream.

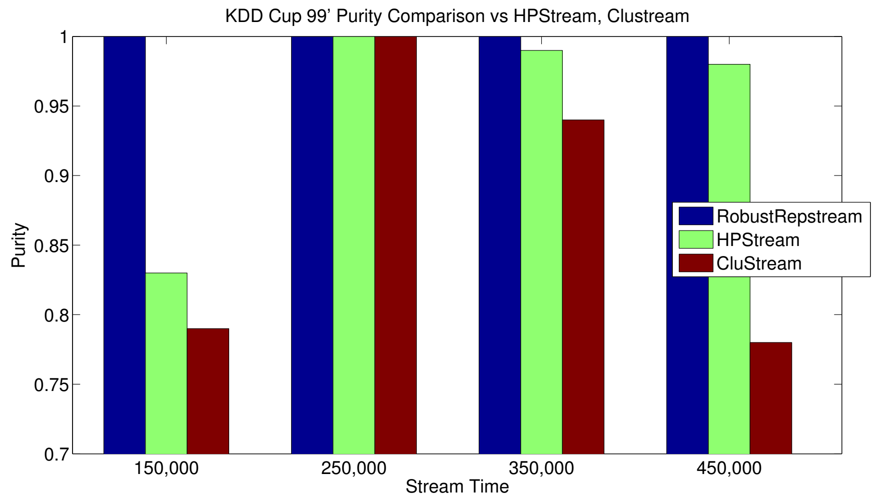


(b) Comparative purity for TreeCov data set against HPStream and CluStream.

Figure 5.12: Comparative purity for TreeCov dataset.



(a) Comparative purity for KDD data against ExCC, STRAP, and HPStream. set



(b) Comparative purity for KDD data against HPStream and CluStream. set

Figure 5.13: Comparative purity for KDD dataset.

dense according to input parameters, which allows D-Stream to effectively find arbitrarily shaped parameters. DBStream, on the other hand, is a sophisticated micro-cluster based approach which seeks to solve weaknesses of previous micro-cluster approaches. DBStream allows micro-clusters to overlap, and records the shared density in these overlapping regions. This additional information allows better clustering decisions to be made, as the shared regions give insight into the arrangement of the underlying data points. Both D-Stream and DBStream are high quality stream clustering approaches that perform well against contemporary algorithms. The stream package for R was used to run these algorithms on the data sets. The data sets were normalised between values of 0 and 1 in each data dimension. For each data set the algorithms were parametrised according to the suggested parameter values in the respective papers and the documentation for their implementations. D-Stream grid-size parameter was set to $len = 0.05$, its dense and sparse cell thresholds set to $C_m = 3.0$ and $C_m = 0.8$, the decay value $\lambda = 0.998$, and its sporadic cell deletion parameter $\beta = 0.3$. The DBStream algorithm was set with its micro-cluster radius $r = 0.05$, its decay parameter $\lambda = 0.01$, its clean-up interval $t\_gap = 1000$, the minimum weight $w\_min = 3.0$, and its intersection factor $\alpha = 0.1$. The purity horizon, as we noted earlier, is 200, and the datasets have been normalised between 0 and 1 in all dimensions.

Figure 5.14(a) shows the comparative purity results for the Closer data set using D-Stream, DBStream, and RobustRepStream. On this data set RobustRepStream outperforms the other algorithms in general over the first and last 10,000 data points, reaching almost 100% purity during these times. during the middle third of the stream, however, DBStream on average has a more consistent higher purity value, possibly due to its shared density feature preventing the overlapping distributions from merging together. RobustRepStream, on the other hand during this period, has periods when its purity value is higher, but also has periods where it achieves a lower purity value than DBStream. As shown in Table 5.1 however, the overall average purity of RepStream is higher than that of DBStream.

Figure 5.14(b) shows the clustering purity for the SynTest data set for the three noted algorithms. On average the three algorithms achieve very similar clustering quality. Table 5.1 shows how the average (mean) purity differs by only 1.8% between the algorithms, achieving 0.969, 0.951, and 0.966 purity for DBStream, D-Stream, and RobustRepStream respectively.

Figure 5.14(c) and Figure 5.14(d) show the clustering results for the DS1 and DS2 data sets respectively. These data sets do not evolve over time or simulate a stream, rather they are static distributions with concave classes, and classes within other classes, making them difficult to cluster. On these data sets the algorithms achieve similar clustering purity. Overall their purity, shown in Table 5.1, differs very little, though with the graph-based D-Stream performing noticeably worse than the other algorithms. RobustRepStream and DBStream differ in purity by less than 0.03 on average.

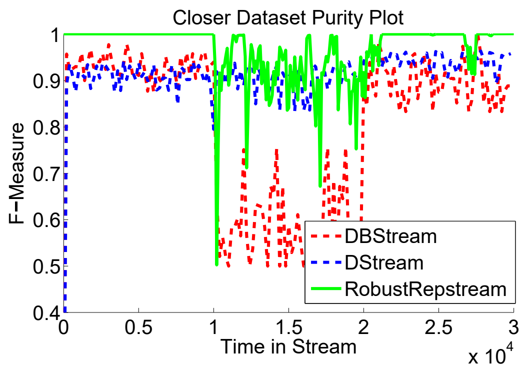
Figure 5.14(e) shows the purity results achieved for the Shapes data set, which simulated the evolution of a stream, by morphing the size of distributions over time as well as combining two distributions together. For this data set DBStream outperforms both D-Stream and RobustRepStream, achieving average purity 2% higher than D-Stream and 0.8% higher than RobustRepStream.

As for the real world benchmark test data sets Figure 5.15 shows the purity of the KDD Cup 99' Network Intrusion data set. For most of the stream all three algorithms achieve perfect purity, due to the presence of only one class for most of the stream. This only differs during the network attack simulations, in which abnormal traffic is inserted into the stream, represented by data points with different class labels mixing in with the typical traffic. During the attacks all algorithms achieve less than perfect purity, with D-Stream performing the worst, having the greatest drops in purity, the average purity for D-Stream over the data set is 0.989. DBStream fares better during the attacks, with an average purity over the whole stream of 0.993. RobustRepStream performs the best on this data set, achieving 0.999 purity on average, and never dropping below 0.9 purity during the length of the stream.

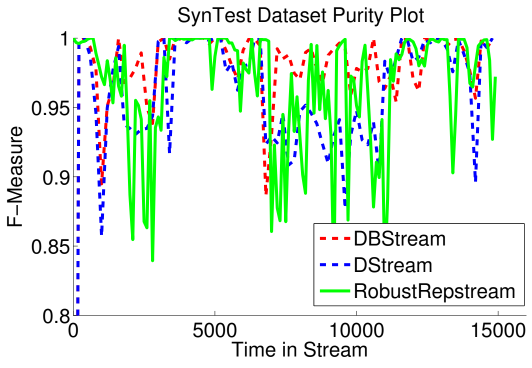
The most stark difference between the clustering algorithms is shown in the Tree Cover Type data set, presented in Figure 5.16. This data set is difficult to cluster, having overlapping classes and 10 different attributes for the data. On average RobustRepStream achieves a purity of 0.884, significantly outperforming both D-Stream and DBStream, which achieve 0.690 and 0.508 purity respectively, as shown in Table 5.1.

### 5.3.5 Baseline for Comparison

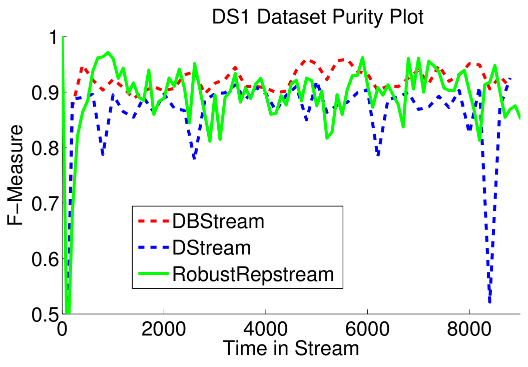
Because RobustRepStream is an extension of the RepStream algorithm it makes sense to compare its performance to the algorithm on which it is based. RepStream, as shown in its original paper (Lühr & Lazarescu, 2009) performs well against other stream clustering algorithms. To compare the relative performance of RobustRepStream compared to RepStream we start by determining what our baseline for comparison is. Since we are extending the RepStream method we want to compare against the best possible



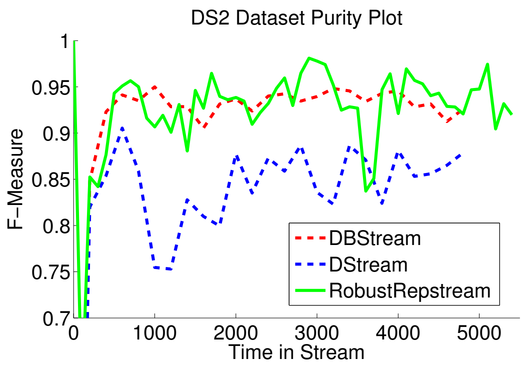
(a) Comparative purity for Closer data set



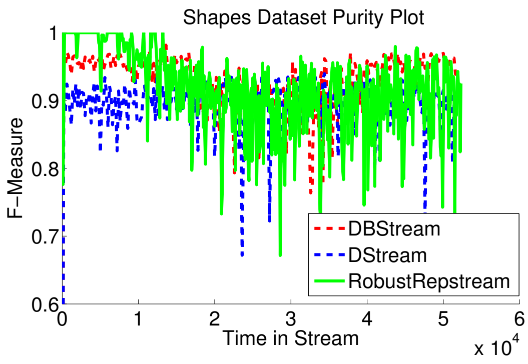
(b) Comparative purity for SynTest data set



(c) Comparative purity for DS1 data set



(d) Comparative purity for DS2 data set



(e) Comparative purity for Shapes data set

Figure 5.14: Comparative purity for our Synthetic datasets against D-Stream and DB-Stream.

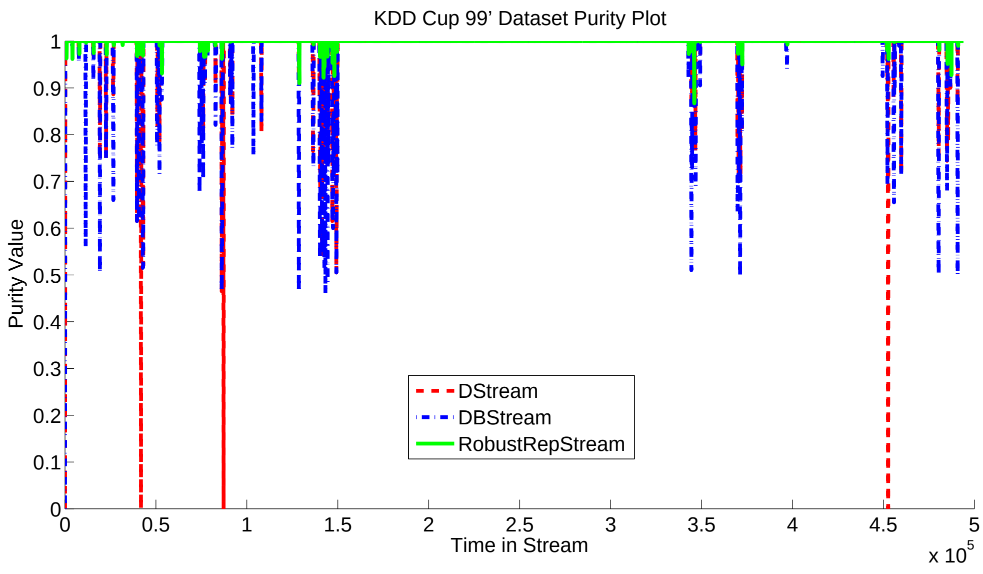


Figure 5.15: Comparative purity for KDD Cup 99' data set

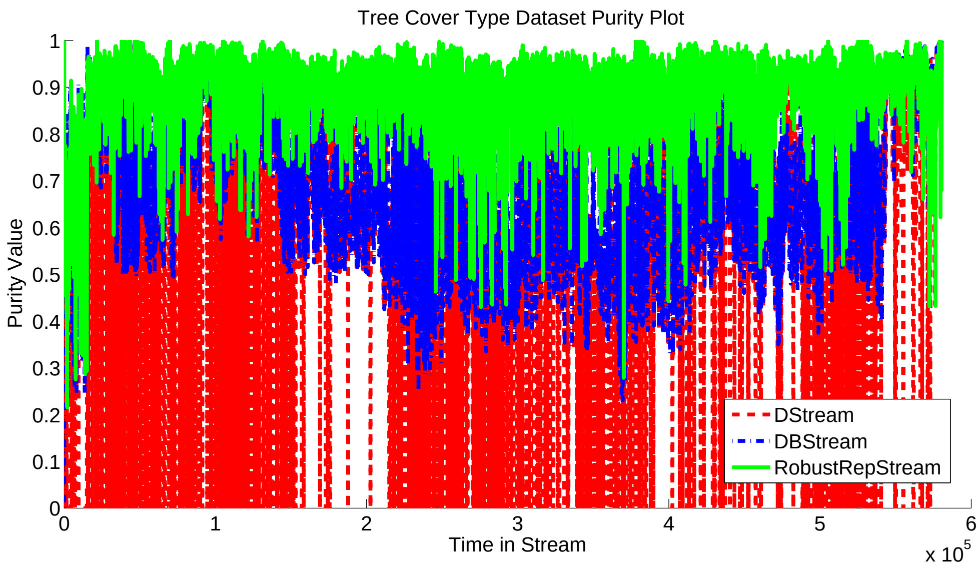


Figure 5.16: Comparative purity for Tree Cover data set

Table 5.1: Average Purity score of DBStream, D-Stream, and RobustRepStream.

| **Data set** | **DBStream** | **D-Stream** | **RobustRepStream** |
|---|---|---|---|
| DS1 | **0.902** | 0.851 | 0.893 |
| DS2 | 0.893 | 0.812 | **0.926** |
| SynTest | **0.969** | 0.951 | 0.966 |
| Closer | 0.799 | 0.913 | **0.968** |
| Shapes | **0.926** | 0.894 | 0.918 |
| TreeCov | 0.508 | 0.690 | **0.884** |
| KDD99 | 0.993 | 0.989 | **0.999** |

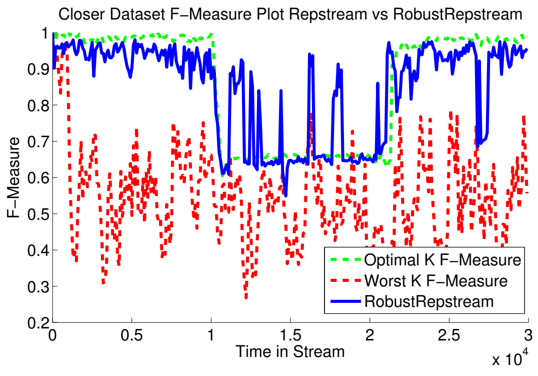
Table 5.2: Best $K$ values

| **Data set** | **Optimal $K$** | **Avg $F$ measure** |
|---|---|---|
| DS1 | 7 | 0.7208 |
| DS2 | 7 | 0.6371 |
| Closer | 9 | 0.8614 |
| SynTest | 9 | 0.7989 |
| Shapes | 6 | 0.6234 |
| KDD | 30 | 0.7898 |
| TreeCov | 29 | 0.6108 |

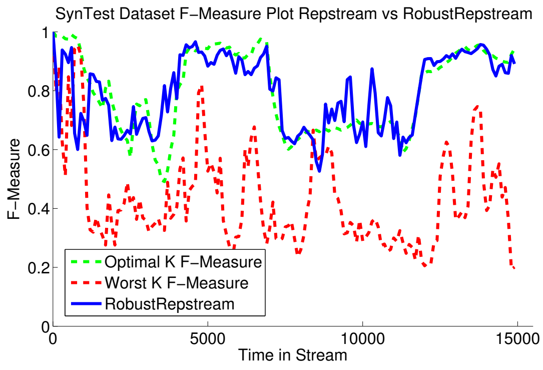
theoretical performance of RepStream. To do this we run RepStream between a range of $K$ values, and determine which $K$ value produces the best overall performance.

To do this we ran multiple instances of RepStream over a range of $K$ values from $K = 5$ to $K = 30$, using $\alpha = 2.0$, and using a memory limit of 1000 points. The reason for this range is that $K$ values lower than 5 tend to fragment clusters into dozens of tiny clusters, while values higher than 30 tend to put all data points into the same cluster. Additionally the vanilla normalisation method was used, with Manhattan distance as the distance metric.

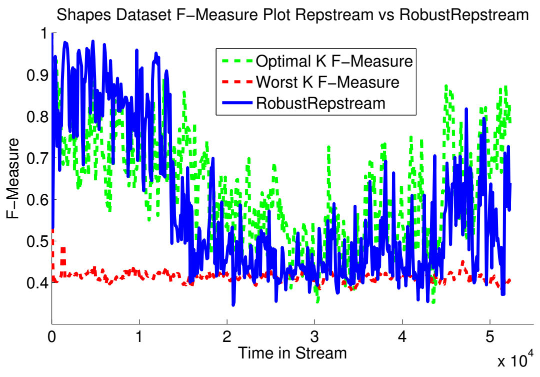
$F$ measure was calculated every 100 data points, from which we compute the mean $F$ measure over the length of the stream. Table 5.2 shows the $K$ value which produced the highest overall $F$ measure for each of our test data sets, and the corresponding $F$ measure values. These $K$ values are the ones for which the original RepStream produced the highest overall $F$ measure scores through the length of the stream. We will evaluate our dynamic variation in comparison to these, and will refer to them as the optimal $K$ values.



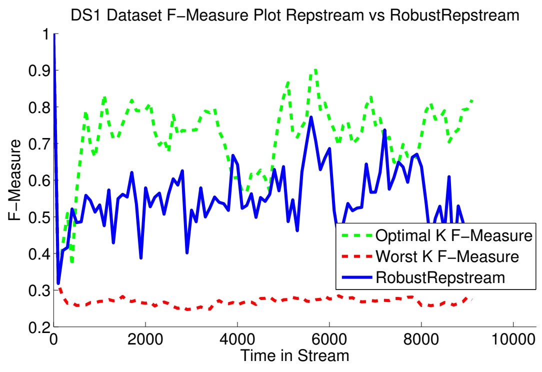
(a) Results for the Closer data set.



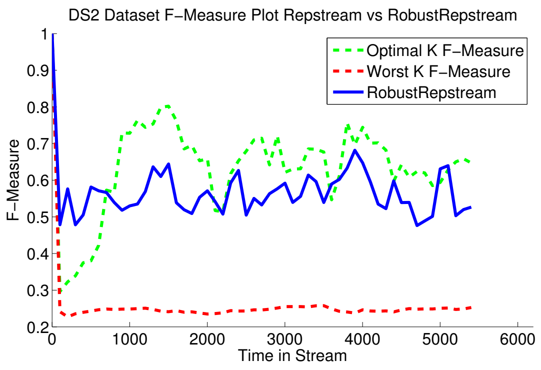
(b) Results for the SynTest data set.



(c) Results for the Shapes data set.



(d) Results for the DS1 data set.



(e) Results for the DS2 data set.

Figure 5.17: F-Measure scores for RobustRepStream versus optimally-parametrised RepStream

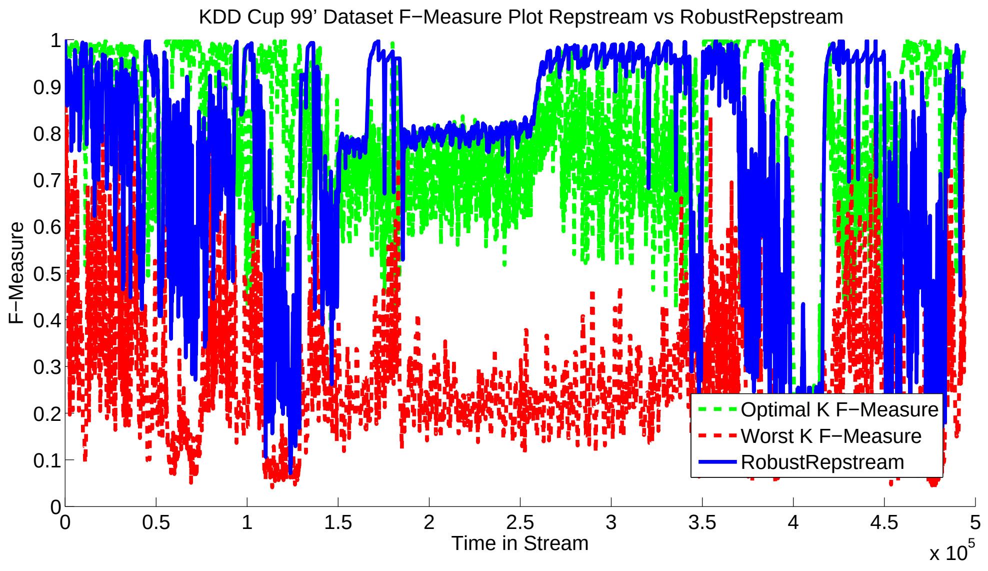


Figure 5.18: Results for the KDD data set.

Table 5.3: Average $F$ measure values of base RepStream with the Best, Worst, and in-between $K$ values, compared to our RobustRepStream method.

| **Data set** | **RobustRepStream** | **Best** $K$ | **Worst** $K$ | **Avg** $K$ |
|---|---|---|---|---|
| Closer | 0.8450 | 0.8614 | 0.5435 | 0.8589 |
| SynTest | 0.7945 | 0.7989 | 0.4345 | 0.7435 |
| KDD | 0.7644 | 0.7898 | 0.2636 | 0.7160 |
| TreeCov | 0.5959 | 0.6108 | 0.2978 | 0.6088 |
| Shapes | 0.5810 | 0.6234 | 0.4140 | 0.4427 |
| DS1 | 0.5532 | 0.7208 | 0.2767 | 0.3134 |
| DS2 | 0.5679 | 0.6371 | 0.2594 | 0.2656 |

### 5.3.6 Evaluation of RobustRepStream

Our dynamic version of RepStream was set to run on our evaluation data sets using an $\alpha$ scaling factor value of 2.0, the same as our evaluation baseline. This parameter is also used as the skewness excess threshold, for the reasons mentioned in Section 5.2.2. The memory limit was set to 1000 points, and used vanilla normalisation, the default value of 0.99 was used for the $\lambda$ decay factor, and Manhattan distance as the distance metric.

Figure 5.17(a) shows the $F$ measure of our method on the Closer data set. The dynamic method, plotted in solid blue, follows even closer to the optimal than on the previous data set. The average $F$ measure of 0.7859 of our method compared to 0.8614 (shown in Table 5.3) for the optimal $k$ leaves an average difference of only

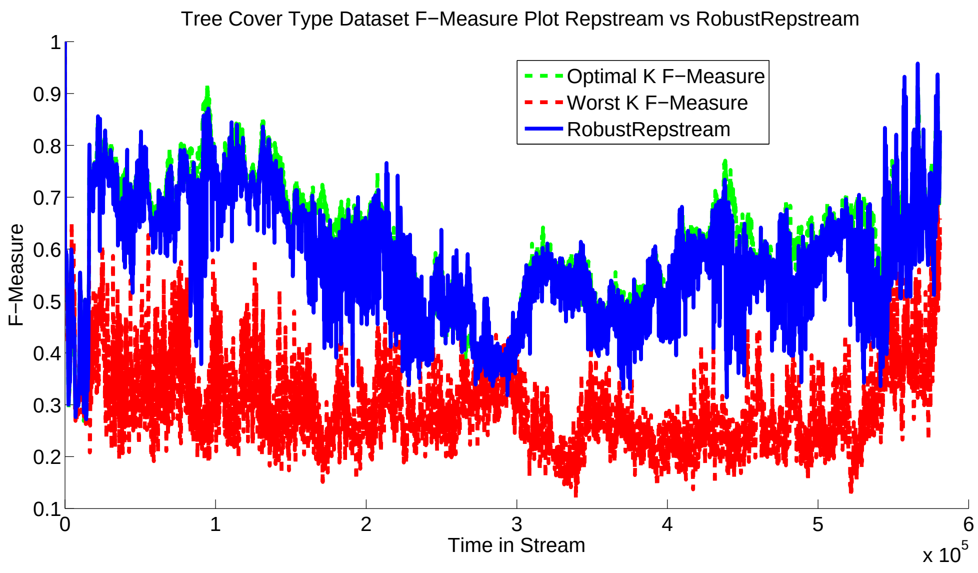


Figure 5.19: Results for the Tree Cover data set.

0.0755. Both the worst and mid $K$ values perform better on this data set in comparison, with the worst $K$ value having an average $F$ measure of 0.5435, and the mid $K$ value producing a $F$ measure of 0.8589. This implies that on this data set there is a larger range of $K$ values which produce good clustering results.

Figure 5.17(b) shows the $F$ measure scores of our method and our comparison baselines on the SynTest data set. Our
method results in a lower $F$ measure than the best $K$ value over the first and last 10,000 points of the data set, however on average performs better during the middle 10,000 point section. The performance of our method is comparable to that of the mid $K$ value, being only 0.015 different on average, as shown in Table 5.3. Even so, our method performs, on average, only 0.0569 lower in terms of $F$ measure compared to the best possible single static $K$ value.

Figure 5.18 shows the $F$ measure of the RobustRepStream method versus the $F$ measure of the best, worst, and mid single static $K$ values on the KDD Cup 99' data set. This is a real-world high dimensional benchmark data set often used in clustering algorithm evaluation, which is significantly different from the previous synthetic data sets. Our dynamic method on average produces an $F$ measure value of 0.7035. The optimal $k$ value produced a $F$ measure score of 0.7898 meaning that our method remains on average within 0.1 of the optimal. There are times when our method performs worse, however on average its performance is comparable to the optimal value, when poorly selected input parameters could result in $F$ measure values as low as 0.2636 on

Table 5.4: Average purity value of RobustRepStream run at different $\alpha$ threshold values.

| **Data set** | $\alpha = 0.5$ | $\alpha = 1$ | $\alpha = 1.5$ | $\alpha = 2$ | $\alpha = 3$ | $\alpha = 4$ | $\alpha = 5$ | $\alpha = 6$ | $\alpha = 7$ |
|---|---|---|---|---|---|---|---|---|---|
| DS1 | 0.994 | 0.990 | 0.973 | 0.885 | 0.605 | 0.410 | 0.386 | 0.365 | 0.363 |
| DS2 | 0.987 | 0.985 | 0.976 | 0.928 | 0.656 | 0.305 | 0.291 | 0.301 | 0.287 |
| SynTest | 0.994 | 0.992 | 0.987 | 0.973 | 0.884 | 0.848 | 0.846 | 0.845 | 0.845 |
| Closer | 0.998 | 0.997 | 0.994 | 0.974 | 0.847 | 0.830 | 0.813 | 0.806 | 0.762 |
| Shapes | 0.998 | 0.997 | 0.976 | 0.930 | 0.642 | 0.515 | 0.485 | 0.480 | 0.473 |
| TreeCov | 0.973 | 0.963 | 0.946 | 0.886 | 0.722 | 0.674 | 0.659 | 0.655 | 0.653 |
| KDD99 | 1.000 | 1.000 | 0.999 | 0.999 | 0.999 | 0.998 | 0.998 | 0.997 | 0.997 |

Table 5.5: Average $F$ Measure value of RobustRepStream run at different $\alpha$ threshold values.

| **Data set** | $\alpha = 0.5$ | $\alpha = 1$ | $\alpha = 1.5$ | $\alpha = 2$ | $\alpha = 3$ | $\alpha = 4$ | $\alpha = 5$ | $\alpha = 6$ | $\alpha = 7$ |
|---|---|---|---|---|---|---|---|---|---|
| DS1 | 0.247 | 0.321 | 0.553 | 0.534 | 0.316 | 0.279 | 0.276 | 0.276 | 0.277 |
| DS2 | 0.241 | 0.298 | 0.417 | 0.579 | 0.319 | 0.261 | 0.260 | 0.260 | 0.259 |
| SynTest | 0.062 | 0.136 | 0.547 | 0.760 | 0.822 | 0.822 | 0.821 | 0.821 | 0.821 |
| Closer | 0.055 | 0.131 | 0.589 | 0.844 | 0.862 | 0.875 | 0.867 | 0.858 | 0.822 |
| Shapes | 0.124 | 0.219 | 0.551 | 0.599 | 0.480 | 0.432 | 0.422 | 0.421 | 0.422 |
| TreeCov | 0.080 | 0.186 | 0.438 | 0.574 | 0.612 | 0.615 | 0.616 | 0.616 | 0.616 |
| KDD99 | 0.232 | 0.292 | 0.591 | 0.675 | 0.797 | 0.739 | 0.700 | 0.841 | 0.817 |

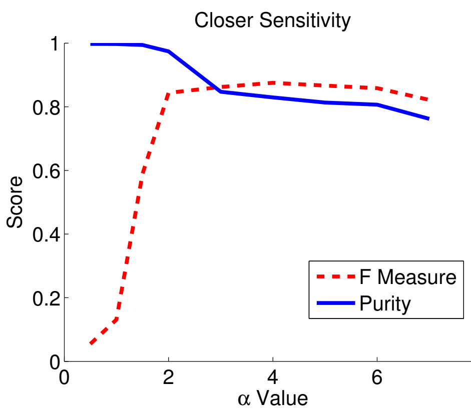
average, as shown in Table 5.3.

Figure 5.19 shows the performance of our method on the Tree Cover Type data set. This data set is significantly more difficult to separate classes, with even the optimal $K$ value having an average $F$ measure score of only 0.6108. Whilst the RobustRepStream method performs comparatively worse on this data set compared to the others, it is still on average only 0.1120 below the optimal, as shown in Table 5.3.

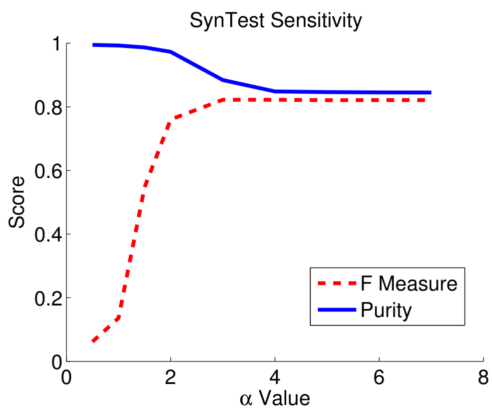
We note that RobustRepStream can still perform almost equivalent to RepStream even when RepStream is supplied with an optimal $K$ value at the beginning of its runtime. However, if optimal parameters cannot be guaranteed, as is to be assumed when dealing with unlabelled data in an unsupervised context, that RobustRepStream outperforms RepStream set with non-optimal parameters.

### 5.3.7 Sensitivity

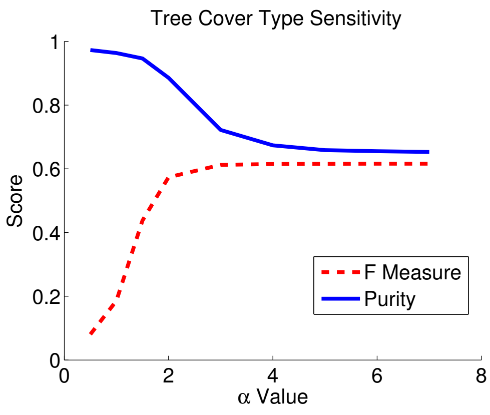
We investigate the sensitivity of the $\alpha$ parameter with regards to clustering quality. Since we remove the $K$ value used by the original RepStream, and replace it with our



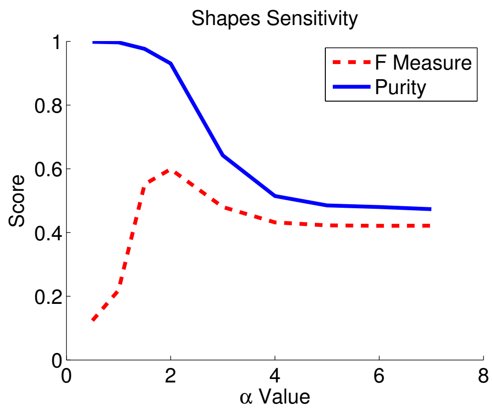
(a) Sensitivity of $\alpha$ for Closer dataset.



(b) Sensitivity of $\alpha$ for SynTest data set.



(c) Sensitivity of $\alpha$ for Tree Cover Type data set.



(d) Sensitivity of $\alpha$ for Shapes data set.

Figure 5.20: Sensitivity of RobustRepStream to the $\alpha$ parameter

edge selection process described in section 5.2.2, we wish to show how sensitive our changes are. Figure 5.20 (a,b,c,d) show both the purity and $F$ measure scores of some of our datasets.

In all cases $\alpha$ values below 2 produce very low $F$ measure values whilst producing very high purity values. This is due to under-clustering, where the algorithm is resulting in too many clusters containing too few points. We expect this result as we described earlier. This can be seen in Table 5.5 and Table 5.4.

At higher values, around $2 \leq \alpha \leq 3$, the $F$ measure increases and plateaus thereafter, whilst the purity value decreases to a roughly stable level. From these sensitivity tests we show that an $\alpha$ value of between 2 and 3 consistently results in the best balance between purity and $F$ measure scores. We note that increases in $F$ measure, which measures both precision and recall, can be achieved with higher $\alpha$ values, however the improvements in the cases of our sensitivity experiments, are marginal.

## 5.4 Results Discussion

In Section 5.3 we compare RobustRepStream against the published results of several other stream clustering algorithms, specifically STRAP, ExCC, HPStream, and the more recent D-Stream and DBStream.

As shown in the comparative purity results in Figure 5.13(a) and 5.12(a), RobustRepStream performs well against the results published for RepStream, ExCC, STRAP, and HPStream. RobustRepStream results in similar clustering quality to RepStream in almost all cases, and outperforms RepStream slightly in purity on the Tree Cover Type data set.

Compared to the algorithms D-Stream and DBStream, RobustRepStream performs very favourably. DBStream marginally outperforms RobustRepStream on the SynTest, DS1, DS2, and Shapes data set very marginally, as shown in Table 5.1, however RobustRepStream performs very favourably compared to these algorithms on the Closer, Tree Cover, and KDD99' data sets. It is worth noting that these results are achieved with all algorithms using the same input parameters for all data sets. Without tuning the parameters RobustRepStream performs comparably to the other algorithms, and outperforms them on the real-world benchmark data sets.

Compared to RepStream, the RobustRepStream algorithm performs well. Even when comparing to the best case, setting RepStream optimally, the RobustRepStream algorithm with no parameter adjustment between data sets has similar performance in terms of $F$ measure to RepStream using the best possible value for the $K$ parameter. This is notable in the 2-dimensional synthetic datasets, as well as the benchmark datasets - Tree Cover and KDD Cup, which have 10 and 43 dimensions respectively, which demonstrates that our method is able to perform on datasets with varying levels of dimensionality.

RobustRepStream is an extension of the RepStream algorithm with the goal of reducing the need for users to set the input parameters, these experiments show that by using the same parameter values for all data sets the RobustRepStream algorithm is able to perform similarly to RepStream even though the RepStream algorithm has the advantage of being tuned for the data set. This situation is unrealistic in the favour of RepStream because one can't expect to know what the best parameter values are before beginning, so this comparison is made to a hypothetical best case, yet still performs well.

In terms of sensitivity the value chosen for the $\alpha$ threshold seems to be insensitive to the data set used. In Section 5.3.7 we show the purity values of RobustRepStream run at different $\alpha$ values on various data sets. In our experiments we used $\alpha = 2$ to represent that each vertex may have one or fewer anomalous outgoing edges, and this value achieves very high quality results in terms of purity. The highest combination of purity and $F$ measure score is achieved with a value between $2 \leq \alpha \leq 3$ in the tested data sets. Whereas the $K$ value of RepStream needed to be adjusted for each data set in order to achieve the highest quality performance, our method provides consistently high quality clustering output without the need to tune our hyper parameters for all the data sets that we tried. This is in line with our goal of removing the need for users to set the sensitive $K$ parameter in RepStream and of making the algorithm more robust.

## 5.5 Conclusion

In this chapter we introduced a clustering algorithm, RobustRepStream, an extension of the RepStream algorithm, that removes the need to set its primary input parameter, the $K$ value, which directly affects the interconnectivity of its graph-based structure. We employ a computed feature called the skewness excess to automatically set the number of outgoing edges for each vertex in the graph automatically and dynamically over time. Our method selects the local connectivity parameter as the maximum value at which the skewness excess is still below a pre-defined universal threshold. This process is performed continuously on new vertices into the graph, and for any nearby vertices for which the new vertex becomes a potential neighbour. Using this method we maintain a non-static and non-uniform number of outgoing edges for vertices in the graph.

We propose that this same approach can be applied in any situation in which a $K$-NN graph would be used. The ability to dynamically set the number of outgoing edges on a per-vertex basis reduces the dependence on knowing an appropriate $K$ value. We further propose that the usage of the skewness excess in determining outgoing edges in a directed nearest neighbour graph can be generalised and used in building graphs in situations where a static $K$ value is not appropriate. This method of selecting edges is not unique to the RobustRepStream algorithm, and could be applied in any nearest neighbour graph-based context. We expect future findings will confirm our conjecture.

Our method successfully removes the need for the user to set the $K$ parameter which is crucial for the success of the original RepStream algorithm. By making use of the $\alpha$ value which is already set by the user as a threshold our RobustRepStream algorithm can achieve performance comparable to that of RepStream when it has an optimally set $K$ value. Given that it is virtually impossible to know what the optimal $K$ value would be for an unlabelled data set, this results in one fewer sensitive parameter which needs to be set for the algorithm. RepStream is shown to be much less sensitive in terms of performance variance for the remaining parameter, the $\alpha$ scaling factor than to the $K$ value. As we show in Section 5.3.7, this parameter is far less sensitive, and results in consistent clustering quality across different datasets in RobustRepStream. This is in contrast to the sensitive nature of the $K$ parameter, which as we show in table 5.2 requires very different values from data set to data set.

We have shown experimentally on average for the data sets that evolve over time that our RobustRepStream performs within an $F$ measure margin of 0.03 of the best possible performance that could be achieved by RepStream when the ground-truth is known. For all of our evaluation data sets, setting the $K$ value non-optimally can lead to $F$ measure differences of more than 0.3 compared to the optimal values. This demonstrates the importance that the $K$ parameter had on RepStream and why removing the need to set the $K$ parameter is so valuable.

# Chapter 6

# Conclusions

Stream cluster analysis is as yet an open field of research, with wide-ranging uses across many domains, including analysis of sensor networks, meteorological data, stock market trends, computer network traffic, customer click streams, phone or other communication records, multimedia data, financial transactions, and observational science data (Silva et al., 2013). Many of these domains are particularly difficult to analyse and model because of the evolving nature of the data. That is, rather than having a static model which needs to be identified, the underlying model instead can change over time.

It is with this problem in mind that stream clustering algorithms are designed to operate. Concept drift is one of the most challenging aspects of stream clustering, and there have been a myriad of different techniques which have been proposed to handle this problem, as we discuss in detail in chapter 2.

Sliding windows and applying fading functions to data structures are popular, common techniques for handling evolution over time (Kaur et al., 2015). These techniques work by discarding older data gradually over time, whilst retaining enough information to add new data points into clusters. This can help with adapting to stream evolution by weighting newer data with more importance.

Whilst these techniques can help the algorithm adjust to changes, the initial parametrisation of algorithms remains one of the greatest challenges to face with stream clustering (Silva et al., 2013). One of the aspects of the challenge is that clustering parameters in algorithms are often data-dependent, and requires prior knowledge about the data set in order to select an appropriate value. This requirement is obviously a huge downside, since clustering by its nature is an exploratory form of data analysis, applied to data sets for which little is known. Another aspect of the challenge is that stream evolution can result in situations where input parameters are appropriate at some times during the stream, and allow the algorithm to yield high quality results, but at other times during the stream the same parameters might lead to low quality clustering output. This is the problem of parameter sensitivity in a concept drifting context.

In this thesis we have sought to address these problems, by improving the robustness of the RepStream algorithm, and proposing our own RobustRepStream algorithm.

## 6.1 Change Detection

In Chapter 3 we presented a method for time series change analysis which we applied to features computed from the geometric features of the data representation structures in the RepStream clustering algorithm.

We proposed a number of computable features which can reflect changes in the underlying data composition and concepts. By examining the cluster count over time, the changes of the $K$ nearest-neighbour edges, the cluster merges and splits, the variation in edge lengths, and the cluster membership of data points over time we establish a varied set of data which we can examine to determine information about the input data stream.

We further proposed a time-series change detection algorithm which we apply to our computed features. This method allows us to identify change points in an arbitrarily dimensional data stream. Using our change detection method on these features we created an algorithm which was capable of detecting change points in the stream concepts over time.

We experimentally showed how our method was competitive with the contemporary PCA-based detection technique described by Qahtan et al. (2015).

## 6.2 Dynamic K Parameter Selection

In Chapter 4 we introduced the concept of the edge distribution score, a feature computed from the distribution of edge lengths in the $K$ nearest neighbour graph structure of RepStream. We showed how the edge lengths have a predictable distribution when a vertex is towards the middle of a region of uniform density. When the computed edge distribution score is higher or lower than expected we can infer that the connectivity level is too high or too low, and thus could affect the quality of clustering.

Using this information we presented an extension to the RepStream algorithm, which computes the edge distribution score over time, and varies the $K$ connectivity parameter over time in response to changes in the data stream.

We evaluated our method on both synthetic and real-world benchmark datasets. In our evaluations we found that even when our method was initially parametrised with the worst possible initial $K$ value, the algorithm was able to quickly locate a more appropriate $K$ value, and produce clustering output which was comparable, in terms of purity and F measure, to the output of RepStream when given optimal initial parameters.

## 6.3 RobustRepStream

Building on the work from prior chapters, we presented our RobustRepStream algorithm in Chapter 5. In this chapter we addressed the problem of parameter sensitivity by designing a more robust algorithm, which requires less knowledge of the data set to set initial parameters, and whose parameters were more robust to changes in data distribution as a result of concept drift.

By analysing the edge lengths of each vertex independently we found that the edge length distributions provided useful information about the local neighbourhood. We proposed the skewness excess measure, which uses information about the tail-end of the distribution to determine when inter-class edges might be created.

Using the skewness excess score we designed a scheme for creating self-arranging nearest neighbour sparse graphs which do not require a universal connectivity parameter - the $K$ parameter used in the original RepStream algorithm. This self-arranging graph scheme used data from the local neighbourhood to select appropriate levels of connectivity for each vertex independently.

In experimental evaluation of our RobustRepStream algorithm we found that it produced high quality clustering across multiple different datasets with no parameter tuning, in situations where the original RepStream algorithm required the $K$ parameter to be tuned significantly between the same datasets to achieve similar quality clustering output.

We showed how our RobustRepStream algorithm successfully produces high quality clustering results with fewer initial input parameters than RepStream, and with less need to vary the input parameter values from dataset to dataset. RobustRepStream represents a more robust algorithm, less sensitive to changes in data distribution, and which is easier for users to parametrise and use in their chosen application. We are confident that employing these methods - examining the local impacts of concept drift on data distribution - will result in more robust clustering approaches, and higher quality clustering in the unpredictable domain of stream data clustering.

## 6.4 Future Work

The application of our method could be used to improve the robustness of other clustering algorithms besides RepStream. As we discussed in Chapter 1, the reliance on user-specified input parameters is problematic in an evolving data stream context, and is considered to be one of the most major challenges facing further research into cluster analysis (Silva et al., 2013).

By applying the techniques we have presented in Chapters 4 and 5, other graph-based clustering algorithms could be extended to no longer require setting a $K$ connectivity parameter. Whilst we applied the technique to extend the RepStream algorithm, the technique doesn't necessarily require any particular aspect of the algorithm outside of its nearest-neighbour sparse graph data representation. As such it can easily be applied to other similar graph-based clustering approaches. For example, our approach could be used to set the connectivity on SNCStream (Barddal et al., 2015), which relies on a nearest-neighbour graph and a $K$ connectivity parameter.

Furthermore, the connectivity selection scheme using skewness excess score from Chapter 5 presents an opportunity to create a framework for the construction of self-organising nearest-neighbour sparse graphs which don't rely on a fixed graph-wide connectivity parameter. This could not only be used to design a new, more robust, clustering approach, but could also be applied to any other application which uses graph-based representations of data in which data similarity, in regards to connectivity, is desirable. As an example, the outlier detection algorithm ODIN (Hautamaki et al., 2004) relies on a $K$ nearest neighbour graph structure, which requires user parametrisation. Algorithms like this, which rely on a fixed $K$ connectivity parameter, could be extended and improved with our self-organising nearest neighbour graph construction method, even outside of the clustering domain.

Lastly, the usage of statistical and geometric analysis of data structures to automatically select parameters, or to remove the need for users to set parameters presents exciting opportunities for algorithm designers. It is desirable when algorithms are less sensitive to changes in the data over time and easier to parametrise. We have shown that our methods can be used to improve the robustness of algorithms, and think that similar techniques on non-graph-based algorithms could also yield positive results.

# Bibliography

M. R. Ackermann, et al. (2012). 'StreamKM++: A clustering algorithm for data streams'. *Journal of Experimental Algorithmics* **17**:2–4.

C. C. Aggarwal (2007). *Data streams: models and algorithms*, vol. 31. Springer Science & Business Media.

C. C. Aggarwal (2015). *Data mining: the textbook.* Springer.

C. C. Aggarwal (2018). 'A survey of stream clustering algorithms'. In *Data Clustering*, pp. 231–258. Chapman and Hall/CRC.

C. C. Aggarwal, et al. (2003). 'A framework for clustering evolving data streams'. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pp. 81–92. VLDB Endowment.

C. C. Aggarwal, et al. (2004). 'A framework for projected clustering of high dimensional data streams'. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pp. 852–863. VLDB Endowment.

A. Amini, et al. (2014). 'On density-based data streams clustering algorithms: A survey'. *Journal of Computer Science and Technology* **29**(1):116–141.

C. Aytekin, et al. (2018). 'Clustering and unsupervised anomaly detection with l 2 normalized deep auto-encoder representations'. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–6. IEEE.

T. R. Bandaragoda, et al. (2014). 'Efficient Anomaly Detection by Isolation Using Nearest Neighbour Ensemble'. In *Proceedings of the 2014 IEEE International Conference on Data Mining Workshop (ICDMW)*, pp. 698–705. IEEE.

J. P. Barddal (2019). ‘Vertical and Horizontal Partitioning in Data Stream Regression Ensembles’. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. IEEE.

J. P. Barddal, et al. (2015). ‘SNCStream: A Social Network-based Data Stream Clustering Algorithm’. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing - SAC ’15*, SAC ’15, pp. 935–940, New York, New York, USA. ACM Press.

M. Basseville, et al. (1993). *Detection of abrupt changes: theory and application*, vol. 104. prentice Hall Englewood Cliffs.

P. Berkhin (2006). ‘A survey of clustering data mining techniques’. In *Grouping multidimensional data*, pp. 25–71. Springer.

V. Bhatnagar, et al. (2014). ‘Clustering data streams using grid-based synopsis’. *Knowledge and Information Systems* **41**(1):127–152.

A. Bifet, et al. (2013). ‘CD-MOA: Change Detection Framework for Massive Online Analysis’. In *Proceedings of the 12th International Symposium, IDA 2013*, pp. 92–103.

J. A. Blackard & D. J. Dean (1999). ‘Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables’. *Computers and Electronics in Agriculture* **24**(3):131 – 151.

A. Botta, et al. (2016). ‘Integration of cloud computing and internet of things: a survey’. *Future Generation Computer Systems* **56**:684–700.

H. E. L. Cagnini & R. C. Barros (2016). ‘PASCAL: An EDA for parameterless shape-independent clustering’. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pp. 3433–3440. IEEE.

C. Camara, et al. (2019). ‘Closed-loop deep brain stimulation based on a stream-clustering system’. *Expert Systems with Applications* **126**:187–199.

F. Cao, et al. (2006). ‘Density-Based Clustering over an Evolving Data Stream with Noise.’. In *Proceedings of the 2006 SIAM International Conference on Data Mining*, pp. 326–337. SIAM.

V. Chandola, et al. (2009). ‘Anomaly detection: A survey’. *ACM Computing Surveys (CSUR)* **41**(3):15.

C. A. Charu & K. Chandan (2013). ‘Data clustering: algorithms and applications’.

Y. Chen & L. Tu (2007). ‘Density-based clustering for real-time stream data’. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD ’07*, p. 133, New York, New York, USA. ACM, ACM Press.

S. Ding, et al. (2016). ‘An Adaptive Density Data Stream Clustering Algorithm’. *Cognitive Computation* **8**(1):30–38.

M. H. Dunham & M. Hahsler (2011). ‘Temporal Structure Learning for Clustering Massive Data Streams in Real-Time’. In *Proceedings of the 2011 SIAM International Conference on Data Mining*, pp. 664–675.

M. Ester, et al. (1996). ‘A density-based algorithm for discovering clusters in large spatial databases with noise.’. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*, vol. 96, pp. 226–231.

A. Forestiero, et al. (2013). ‘A single pass algorithm for clustering evolving data streams based on swarm intelligence’. *Data Mining and Knowledge Discovery* **26**(1):1–26.

J. Forrest (2011). ‘Stream: A framework for data stream modeling in R’. *Bachelor Thesis, Department of Computer Science and Engineering, SMU* .

A. Foss & O. R. Zaiane (2002). ‘A parameterless method for efficiently discovering clusters of arbitrary shape in large datasets’. In *Proceedings of the 2002 IEEE International Conference on Data Mining, ICDM 2003*, pp. 179–186.

J. Gama, et al. (2014). ‘A survey on concept drift adaptation’. *ACM Computing Surveys (CSUR)* **46**(4):44.

J. Gao, et al. (2007). ‘A general framework for mining concept-drifting data streams with skewed distributions’. In *Proceedings of the 2007 Siam International Conference on Data Mining*, pp. 3–14. SIAM.

J. Gao, et al. (2005). ‘An Incremental Data Stream Clustering Algorithm Based on Dense Units Detection’. In T. B. Ho, D. Cheung, & H. Liu (eds.), *Proceedings* *of the 9th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, pp. 420–425, Hanoi, Vietnam. Springer.

F. Gorunescu (2011). *Data Mining: Concepts, models and techniques*, vol. 12. Springer Science & Business Media.

F. Gouineau, et al. (2016). 'PatchWork, a scalable density-grid clustering algorithm'. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing - SAC '16*, SAC '16, pp. 824–831, New York, New York, USA. ACM Press.

S. Guha, et al. (2000). 'Clustering Data Streams'. In *Proceedings of the Annual Symposium on Foundations of Computer Science*, pp. 359–366.

F. Gustafsson & F. Gustafsson (2000). *Adaptive filtering and change detection*, vol. 1. Citeseer.

M. Hahsler & M. Bolaos (2016). 'Clustering Data Streams Based on Shared Density between Micro-Clusters'. *IEEE Transactions on Knowledge and Data Engineering* **28**(6):1449–1461.

M. Hassani, et al. (2014). 'Adaptive Multiple-Resolution Stream Clustering'. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 134–148. IEEE.

V. Hautamaki, et al. (2004). 'Outlier detection using k-nearest neighbour graph'. In *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.*, vol. 3, pp. 430–433. IEEE.

S. Hettich & S. D. Bay (1999). 'The UCI KDD Archive [http://kdd.ics.uci.edu]'.

G. Hulten, et al. (2001). 'Mining time-changing data streams'. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 97–106. ACM.

M. Imran, et al. (2018). 'Processing social media messages in mass emergency: Survey summary'. In *Companion Proceedings of the The Web Conference 2018*, pp. 507–511. International World Wide Web Conferences Steering Committee.

A. K. Jain, et al. (1999). 'Data clustering: a review'. *ACM Computing Surveys (CSUR)* **31**(3):264–323.

J. H. Janssens, et al. (2009). 'Outlier detection with one-class classifiers from ML and KDD'. In *Proceedings of the International Conference on Machine Learning and Applications*, pp. 147–153. IEEE.

S. Kaur, et al. (2015). 'Stream Clustering Algorithms: A Primer'. In *Big Data in Complex Systems*, vol. 9, pp. 105–145. Springer.

M. Khalilian & N. Mustapha (2010). 'Data stream clustering: Challenges and issues'. *arXiv preprint arXiv:1006.5261* .

I. Khamassi, et al. (2018). 'Discussion and review on evolving data streams and concept drift adapting'. *Evolving Systems* **9**(1):1–23.

D. Kifer, et al. (2004). 'Detecting change in data streams'. In *Proceedings of the 13th International Conference on Very large Data Bases*, pp. 180–191. VLDB Endowment.

H. Kremer, et al. (2011). 'An effective evaluation measure for clustering on evolving data streams'. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 868–876. ACM.

A. Kumar et al. (2017). *Energy Efficient Clustering Algorithm for Wireless Sensor Network*. Ph.D. thesis, Lovely Professional University.

G. H. Lee (2016). 'Grid-based dynamic clustering with grid proximity measure'. *Intelligent Data Analysis* **20**(4):853–875.

F. Li, et al. (2018). 'Discriminatively boosted image clustering with fully convolutional auto-encoders'. *Pattern Recognition* **83**:161–173.

R. Logesh, et al. (2019). 'Enhancing recommendation stability of collaborative filtering recommender system through bio-inspired clustering ensemble method'. *Neural Computing and Applications* pp. 1–24.

S. Lühr & M. Lazarescu (2009). 'Incremental clustering of dynamic data streams using connectivity based representative points'. *Data & Knowledge Engineering* **68**(1):1–27.

J. MacQueen et al. (1967). 'Some methods for classification and analysis of multivariate observations'. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, pp. 281–297. Oakland, CA, USA.

M. Maier, et al. (2007). ‘Cluster identification in nearest-neighbor graphs’. In *Proceedings of the 18th International Conference on Algorithmic Learning Theory*, pp. 196–210. Springer.

J. Mao & A. K. Jain (1996). ‘A self-organizing network for hyperellipsoidal clustering (HEC)’. *IEEE Transactions on Neural Networks* **7**(1):16–29.

M. Masud, et al. (2011). ‘Classification and Novel Class Detection in Concept-Drifting Data Streams under Time Constraints’. *IEEE Transactions on Knowledge and Data Engineering* **23**:859–874.

W. M. B. W. Mohd, et al. (2012). ‘An improved parameter less data clustering technique based on maximum distance of data and Lioyd k-means algorithm’. *Procedia Technology* **1**:367–371.

O. Nasraoui & C. Rojas (2006). ‘Robust Clustering for Tracking Noisy Evolving Data Streams’. In *Proceedings of the 2006 SIAM International Conference on Data Mining*, pp. 619–623. SIAM.

L. O’callaghan, et al. (2002). ‘Streaming-data algorithms for high-quality clustering’. In *Proceedings of the 18th International Conference on Data Engineering*, p. 0685. IEEE.

D.-S. Pham, et al. (2014). ‘Anomaly detection in large-scale data stream networks’. *Data Mining and Knowledge Discovery* **28**(1):145–189.

A. A. Qahtan, et al. (2015). ‘A PCA-Based Change Detection Framework for Multidimensional Data Streams: Change Detection in Multidimensional Data Streams’. In *Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 935–944. ACM.

S. Rajasegarar, et al. (2014). ‘Hyperspherical cluster based distributed anomaly detection in wireless sensor networks’. *Journal of Parallel and Distributed Computing* **74**(1):1833–1847.

P. J. Rousseeuw & C. Croux (1993). ‘Alternatives to the Median Absolute Deviation’. *Journal of the American Statistical Association* **88**(424):1273–1283.

C. Ruiz, et al. (2009). ‘C-DenStream: Using domain knowledge on a data stream’. In *Proceedings of the 12th International Conference on Discovery Science*, pp. 287–301. Springer.

J. C. Schlimmer & R. H. Granger (1986). ‘Incremental learning from noisy data’. *Machine learning* **1**(3):317–354.

J. Schneider & M. Vlachos (2013). ‘Fast parameterless density-based clustering via random projections’. In *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management - CIKM ’13*, pp. 861–866, New York, New York, USA. ACM, ACM Press.

J. Shao, et al. (2019). ‘Synchronization-based clustering on evolving data stream’. *Information Sciences* **501**:573–587.

J. A. Silva, et al. (2013). ‘Data stream clustering: A survey’. *ACM Computing Surveys (CSUR)* **46**(1):13.

A. Soule, et al. (2005). ‘Combining filtering and statistical methods for anomaly detection’. In *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement*, pp. 31–31. USENIX Association.

S. J. Stolfo, et al. (2000). ‘Cost-based modeling for fraud and intrusion detection: Results from the JAM project’. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, vol. 2, pp. 130–144. IEEE.

L. Sun, et al. (2017). ‘Fast affinity propagation clustering based on incomplete similarity matrix’. *Knowledge and Information Systems* **51**(3):941–963.

C. Truong, et al. (2018). ‘A review of change point detection methods’. *arXiv preprint arXiv:1801.00718* .

V. S. Tseng & C.-P. Kao (2005). ‘Efficiently mining gene expression data via a novel parameterless clustering method’. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* **2**(4):355–365.

M. Van Leeuwen & A. Siebes (2008). ‘Streamkrimp: Detecting change in data streams’. In *Machine Learning and Knowledge Discovery in Databases*, pp. 672–687. Springer.

L. Wan, et al. (2009). ‘Density-based clustering of data streams at multiple resolutions’. *ACM Transactions on Knowledge Discovery from Data* **3**(3):1–28.

G. Wang, et al. (2017). ‘Clickstream user behavior models’. *ACM Transactions on the Web (TWEB)* **11**(4):1–37.

Z. Wang, et al. (2019). ‘A new method for rapid genome classification, clustering, visualization, and novel taxa discovery from metagenome’. *BioRxiv* p. 812917.

N. Wattanakitrungroj, et al. (2018). ‘BEstream: Batch capturing with elliptic function for one-pass data stream clustering’. *Data & Knowledge Engineering* .

G. Widmer & M. Kubat (1996). ‘Learning in the presence of concept drift and hidden contexts’. *Machine learning* **23**(1):69–101.

X. Zhang, et al. (2008). ‘Data streaming with affinity propagation’. In *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 628–643. Springer.

K. Zhao, et al. (2016). ‘Urban human mobility data mining: An overview’. In *2016 IEEE International Conference on Big Data (Big Data)*, pp. 1911–1920. IEEE.

A. Zhou, et al. (2008). ‘Tracking clusters in evolving data streams over sliding windows’. *Knowledge and Information Systems* **15**(2):181–214.

M. Zwolenski, et al. (2014). ‘The digital universe: Rich data and the increasing value of the internet of things’. *Australian Journal of Telecommunications and the Digital Economy* **2**(3):47.

# Statement of Contribution by Others

To Whom It May Concern, I, Ross Callister, contributed to the design and conception of the research ideas, the literature review, the implementation of algorithms, the design and execution of experiments, the collation of experimental results, the paper writing and typesetting, and the discussion of results to the following papers:

- Callister, R., Lazarescu, M., & Pham, D. S. (2015). Detection of Structural Changes in Data Streams. In *Proceedings of the Thirteenth Australasian Data Mining Conference (AusDM 2015), Sydney, Australia, August 2015* (pp. 79-88).
- Callister, R., Lazarescu, M., & Pham, D. S. (2017, April). Graph-based clustering with DRepStream. In *Proceedings of the Symposium on Applied Computing* (pp. 850-857).
- Callister, R., Pham, D. S., & Lazarescu, M. (2019). Using distribution analysis for parameter selection in RepStream. *Mathematical Foundations of Computing*, 2(3), 215-250.
- Callister, R., Pham, D. S., & Lazarescu, M. (2020). RobustRepStream: Robust Stream Clustering Using Self-Controlled Connectivity Graph. *Intelligent Data Analysis*, 24(4).

Ross Callister ...............................

I, as a Co-Author, endorse that this level of contribution by the candidate indicated above is appropriate.

Dr. Duc-Son Pham...............................

Dr. Mihai Lazarescu.............................

# Index

## Copyright Statement

Content from the following papers were used in the production of the thesis titled “Automatically Selecting Parameters for Graph-Based Clustering”, by Ross Callister

**Article 1**
Callister, R., Lazarescu, M., & Pham, D. S. (2015). Detection of Structural Changes in Data Streams. In *Proceedings of the Thirteenth Australasian Data Mining Conference (AusDM 2015)*, Sydney, Australia, August 2015 (pp. 79-88).

Published by the Australian Computer Society Inc, copyright owned by *Conferences in Research and Practice in Information Technology Series (CRPIT)*
The agreement allows material to be used in a thesis provided the given copyright notice is included (which has been done at the start of Chapter 3)

**Article 2**
Callister, R., Pham, D. S., & Lazarescu, M. (2019). Using distribution analysis for parameter selection in RepStream. *Mathematical Foundations of Computing*, 2(3), 215-250.

Published by *The American Institute of Mathematical Sciences*, copyright owned by same.
The agreement allows the use of copyrighted materials in derivative and scholarly works.

**Article 3**
Callister, R., Pham, D. S., & Lazarescu, M. (2020). RobustRepStream: Robust Stream Clustering Using Self-Controlled Connectivity Graph. *Intelligent Data Analysis*, 24(4).

Published by *IOS Press* under licence. Copyright is retained by Ross Callister.
The agreement includes permission to use the article, in whole or in part, as a basis for scholarly works.

**Article 4**
Callister, R., Lazarescu, M., & Pham, D. S. (2017, April). Graph-based clustering with DRepStream. In *Proceedings of the Symposium on Applied Computing* (pp. 850-857).

Published by the Association for Computing Machinery, copyright owned by same.
The agreement allows for the reuse of any portion of the work in future works.

The following pages list the copyright agreements which permit usage of the listed works to be used in the publication of a thesis.

# Article 1, CRPITA copyright agreement

**FAX TO +61 2 9685 9557   ATTENTION: S. SIMOFF**

CONFERENCES IN RESEARCH AND PRACTICE IN INFORMATION TECHNOLOGY
**- Acceptance and Copyright Transfer Form**

**Acceptance**
I/We hereby consent to the publication of my/our paper entitled:

______________________________________________________________

which appeared/will appear at:

______________________________________________________________

and in the Australian Computer Society series *Conferences in Research and the Practice in Information Technology,* Volume____________________

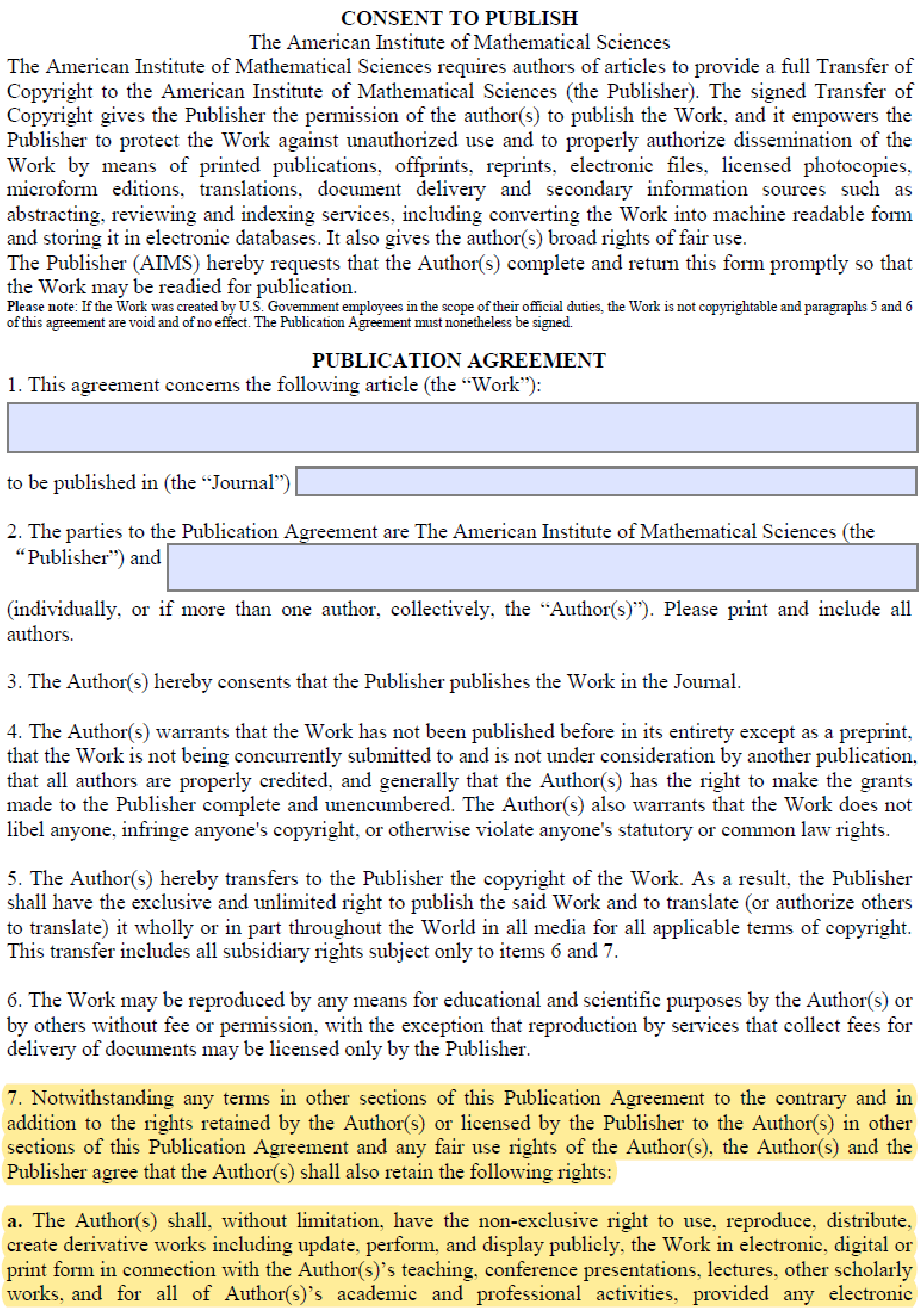
**Assignment in Copyright – Standard Assignment**
We hereby assign all copyright in and to the above work to the Australian Computer Society Inc. (the "ACS"). I/We hereby warrant that the work is original and that I/We am/are the author(s) of the work, except possibly for material such as text passages, figures, and data that clearly identify the original source. I/We have the power and authority to make and execute this assignment. I/We also assert that this material is not libelous and that the publication of this material is not illegal.
In consideration of this assignment the ACS grants to the above authors and employers for whom the work described in the paper may have been performed a royalty-free license to use the paper on the following conditions.

1. Employers (or authors) retail all proprietary rights in any process, procedure, or article of manufacture described in the work.
2. Authors/employers may reproduce or authorize others to reproduce the material extracted verbatim from the paper, or derivative works for the author's personal use or for company use provided that the source and the ACS copyright notice is indicated, that the copies are not used in any way that implies ACS endorsement of a produce or service of an employer, and that the copies themselves are not offered for sale.
3. Authors/employers may make limited distribution of all or portions of the paper prior to publication if they inform the ACS of the nature and extent of such limited distribution prior thereto.
4. In the case of work performed under an Australian or U.S. Government contract or grant, ACS recognizes that the Australian or U.S. Government has royalty-free permission to reproduce all or portions of the above work, and to authorize others to do so, for official Australian or U.S. Government purposes only, if the contract/grant so demands.
5. For all circumstances not covered above, authors/employers must request permission from the ACS.

Please note that, although authors are permitted to use all or portions of their ACS-copyright material in other works, this does not include granting third party requests for reprinting, republishing or other types of reuse. The ACS must handle all third party requests.
Signed and dated:

________________________________________          _________/_________/20__

## Article 2, consent to publish



**CONSENT TO PUBLISH**

The American Institute of Mathematical Sciences

The American Institute of Mathematical Sciences requires authors of articles to provide a full Transfer of Copyright to the American Institute of Mathematical Sciences (the Publisher). The signed Transfer of Copyright gives the Publisher the permission of the author(s) to publish the Work, and it empowers the Publisher to protect the Work against unauthorized use and to properly authorize dissemination of the Work by means of printed publications, offprints, reprints, electronic files, licensed photocopies, microform editions, translations, document delivery and secondary information sources such as abstracting, reviewing and indexing services, including converting the Work into machine readable form and storing it in electronic databases. It also gives the author(s) broad rights of fair use.

The Publisher (AIMS) hereby requests that the Author(s) complete and return this form promptly so that the Work may be readied for publication.

**Please note:** If the Work was created by U.S. Government employees in the scope of their official duties, the Work is not copyrightable and paragraphs 5 and 6 of this agreement are void and of no effect. The Publication Agreement must nonetheless be signed.

**PUBLICATION AGREEMENT**

1. This agreement concerns the following article (the "Work"):

to be published in (the "Journal")

2. The parties to the Publication Agreement are The American Institute of Mathematical Sciences (the "Publisher") and

(individually, or if more than one author, collectively, the "Author(s)"). Please print and include all authors.

3. The Author(s) hereby consents that the Publisher publishes the Work in the Journal.

4. The Author(s) warrants that the Work has not been published before in its entirety except as a preprint, that the Work is not being concurrently submitted to and is not under consideration by another publication, that all authors are properly credited, and generally that the Author(s) has the right to make the grants made to the Publisher complete and unencumbered. The Author(s) also warrants that the Work does not libel anyone, infringe anyone's copyright, or otherwise violate anyone's statutory or common law rights.

5. The Author(s) hereby transfers to the Publisher the copyright of the Work. As a result, the Publisher shall have the exclusive and unlimited right to publish the said Work and to translate (or authorize others to translate) it wholly or in part throughout the World in all media for all applicable terms of copyright. This transfer includes all subsidiary rights subject only to items 6 and 7.

6. The Work may be reproduced by any means for educational and scientific purposes by the Author(s) or by others without fee or permission, with the exception that reproduction by services that collect fees for delivery of documents may be licensed only by the Publisher.

7. Notwithstanding any terms in other sections of this Publication Agreement to the contrary and in addition to the rights retained by the Author(s) or licensed by the Publisher to the Author(s) in other sections of this Publication Agreement and any fair use rights of the Author(s), the Author(s) and the Publisher agree that the Author(s) shall also retain the following rights:

**a.** The Author(s) shall, without limitation, have the non-exclusive right to use, reproduce, distribute, create derivative works including update, perform, and display publicly, the Work in electronic, digital or print form in connection with the Author(s)'s teaching, conference presentations, lectures, other scholarly works, and for all of Author(s)'s academic and professional activities, provided any electronic reproduction faithfully renders the appearance and functionality of each page in its entirety exactly as published online in the Journal.

**b.** Once the Work has been published by the Publisher, the Author(s) shall also have all the non-exclusive rights necessary to make, or to authorize others to make, the text and other content of the Work available in digital form on one or more digital repositories or websites under the control of the Author(s) or a nonprofit entity, provided any electronic reproduction faithfully renders the appearance and functionality of each page in its entirety exactly as published online in the Journal.

**c.** The Author(s) further retains all non-exclusive rights necessary to grant to the Author(s)'s current or future employing institution(s) the non-exclusive right to use, reproduce, distribute, display, publicly perform, and make copies of the Work in electronic, digital or in print form in connection with teaching, digital repositories, conference presentations, lectures, other scholarly works, and all academic and professional activities conducted at the Author(s)'s employing institution(s), provided any electronic reproduction faithfully renders the appearance and functionality of each page in its entirety exactly as published online in the Journal.

**d.** The Author(s) shall have the non-exclusive right to grant permission to other publishers or institutions to publish the Work in an anthology of works on related topics, provided any such publication faithfully reproduces each page in its entirety exactly as it appeared in the Journal.

8. The parties agree that wherever there is any conflict between stated policies of the Publisher and this Publication Agreement, the provisions of this Publication Agreement are paramount, and the Publisher's policies shall be construed accordingly.

9. In the event of receiving any request to reprint or translate all or part of the Work, the Publisher shall seek to inform the Author(s).

10. The Author(s) and the Publisher hereby dedicate the Work to the public domain after 28 years from the date of publication. Works in the public domain are not protected by copyright and can be used freely by everyone.

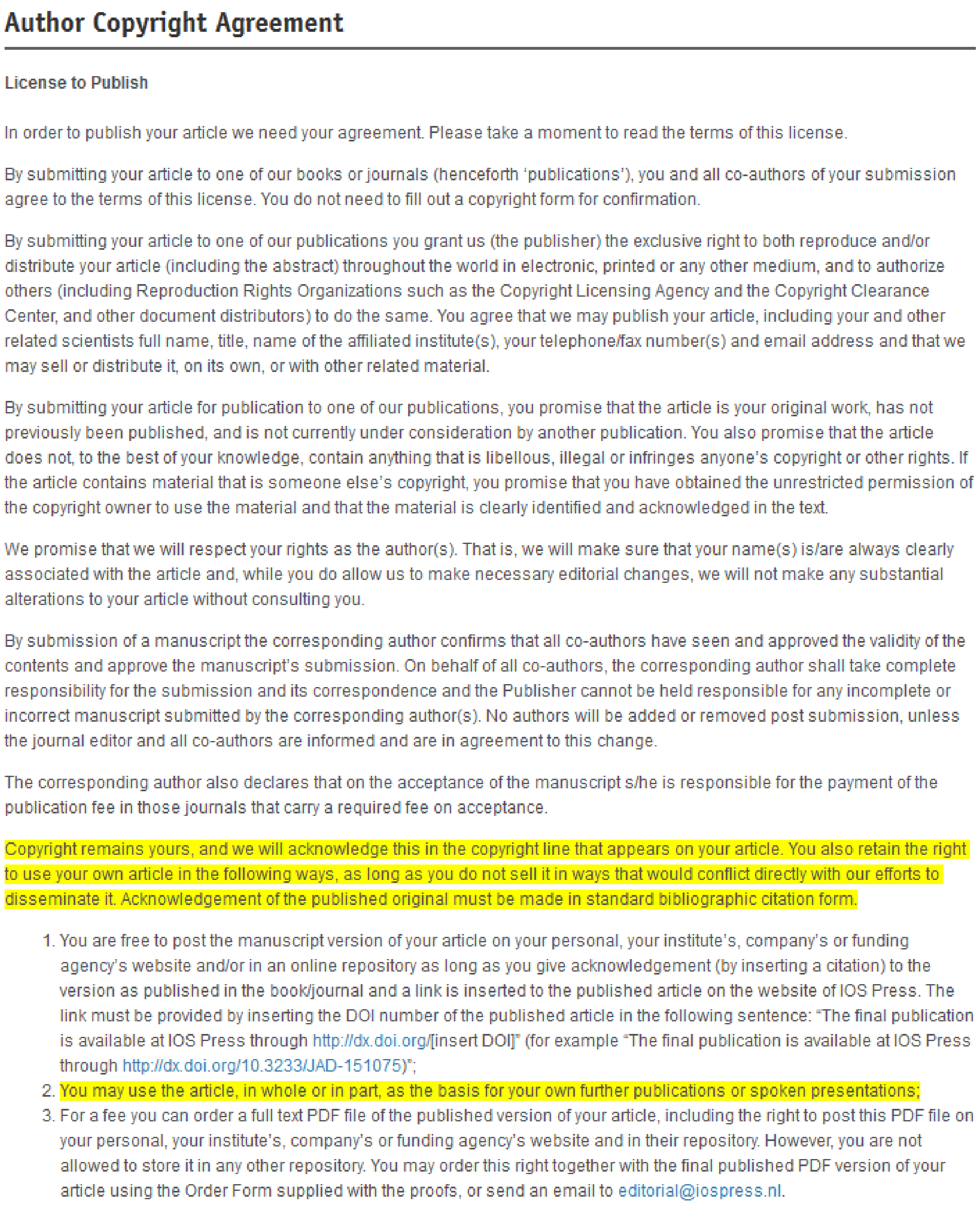
11. This agreement is to be signed by the Author(s) or, in the case of a "work-made-for-hire," by the employer. If there is more than one author, then either all must sign the Publication Agreement, or one author may sign for all provided the signer appends a statement that attests that each author has approved this agreement and has agreed to be bound by it. This Agreement will be governed by the domestic laws of Missouri and will be binding on, and inure to the benefit of, the Author(s)'s heirs and personal representatives and the Publishers successors and assigns.

12. Final Agreement. This Publication Agreement constitutes the final agreement between the Author(s) and the Publisher with respect to the publication of the Work and allocation of rights under copyright in the Work. Any modification of or additions to the terms of this Publication Agreement must be in writing and executed by both Publisher and Author(s) in order to be effective.

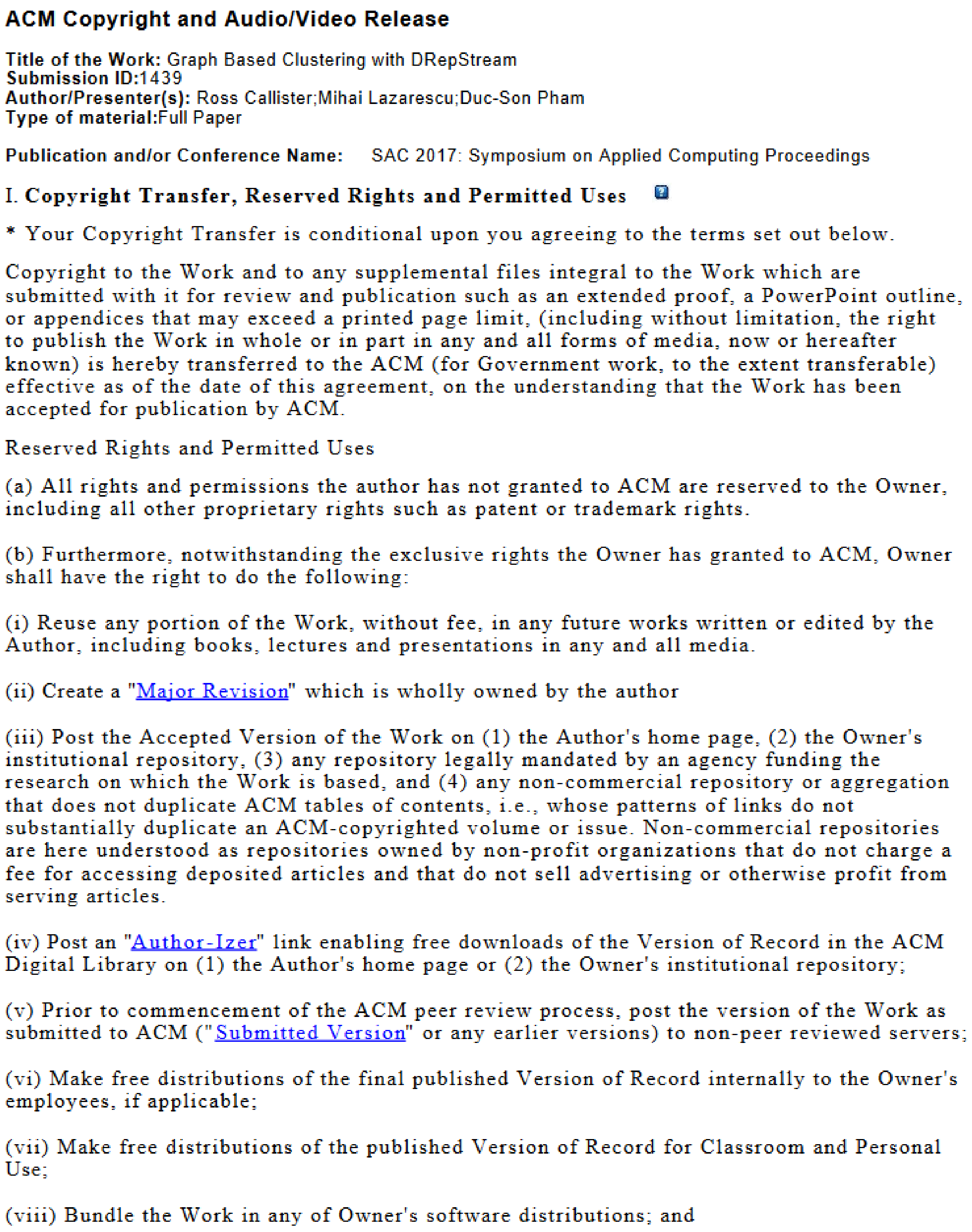
**All authors must sign, or one author may sign if the signer appends a statement that attests that each author has approved this agreement and has agreed to be bound by it.**

| | | |
|---|---|---|
| **Author 1** | **Print name** | |
| | **Signature** | |
| | **Date** | |

## Article 3, copyright agreement



# Article 4, copyright transfer



**ACM Copyright and Audio/Video Release**

**Title of the Work:** Graph Based Clustering with DRepStream
**Submission ID:**1439
**Author/Presenter(s):** Ross Callister;Mihai Lazarescu;Duc-Son Pham
**Type of material:**Full Paper

**Publication and/or Conference Name:** SAC 2017: Symposium on Applied Computing Proceedings

I. **Copyright Transfer, Reserved Rights and Permitted Uses**

* Your Copyright Transfer is conditional upon you agreeing to the terms set out below.

Copyright to the Work and to any supplemental files integral to the Work which are submitted with it for review and publication such as an extended proof, a PowerPoint outline, or appendices that may exceed a printed page limit, (including without limitation, the right to publish the Work in whole or in part in any and all forms of media, now or hereafter known) is hereby transferred to the ACM (for Government work, to the extent transferable) effective as of the date of this agreement, on the understanding that the Work has been accepted for publication by ACM.

Reserved Rights and Permitted Uses

(a) All rights and permissions the author has not granted to ACM are reserved to the Owner, including all other proprietary rights such as patent or trademark rights.

(b) Furthermore, notwithstanding the exclusive rights the Owner has granted to ACM, Owner shall have the right to do the following:

(i) Reuse any portion of the Work, without fee, in any future works written or edited by the Author, including books, lectures and presentations in any and all media.

(ii) Create a "Major Revision" which is wholly owned by the author

(iii) Post the Accepted Version of the Work on (1) the Author's home page, (2) the Owner's institutional repository, (3) any repository legally mandated by an agency funding the research on which the Work is based, and (4) any non-commercial repository or aggregation that does not duplicate ACM tables of contents, i.e., whose patterns of links do not substantially duplicate an ACM-copyrighted volume or issue. Non-commercial repositories are here understood as repositories owned by non-profit organizations that do not charge a fee for accessing deposited articles and that do not sell advertising or otherwise profit from serving articles.

(iv) Post an "Author-Izer" link enabling free downloads of the Version of Record in the ACM Digital Library on (1) the Author's home page or (2) the Owner's institutional repository;

(v) Prior to commencement of the ACM peer review process, post the version of the Work as submitted to ACM ("Submitted Version" or any earlier versions) to non-peer reviewed servers;

(vi) Make free distributions of the final published Version of Record internally to the Owner's employees, if applicable;

(vii) Make free distributions of the published Version of Record for Classroom and Personal Use;

(viii) Bundle the Work in any of Owner's software distributions; and

(ix) Use any Auxiliary Material independent from the Work.

When preparing your paper for submission using the ACM TeX templates, the rights and permissions information and the bibliographic strip must appear on the lower left hand portion of the first page.

The new Authorized ACM TeX template .cls version 2.8, automatically creates and positions these text blocks for you based on the code snippet which is system-generated based on your rights management choice and this particular conference.

*Please copy and paste the following code snippet into your TeX file between \begin{document} and \maketitle, either after or before CCS codes.*

```
\CopyrightYear{2017}
\setcopyright{acmcopyright}
\conferenceinfo{SAC 2017,}{April 03-07, 2017, Marrakech, Morocco}
\isbn{978-1-4503-4486-9/17/04}\acmPrice{\$15.00}
\doi{http://dx.doi.org/10.1145/3019612.3019672}
```

*If you are using the ACM Microsoft Word template, or still using an older version of the ACM TeX template, or the current versions of the ACM SIGCHI, SIGGRAPH, or SIGPLAN TeX templates, you must copy and paste the following text block into your document as per the instructions provided with the templates you are using:*

*NOTE: Make sure to include your article's DOI as part of the bibstrip data; DOIs will be registered and become active shortly after publication in the ACM Digital Library*

[✓] A. Assent to Assignment. I hereby represent and warrant that I am the sole owner (or authorized agent of the copyright owner(s)), with the exception of third party materials detailed in section III below. I have obtained permission for any third-party material included in the Work.

☐ B. Declaration for Government Work. I am an employee of the National Government of my country and my Government claims rights to this work, or it is not copyrightable (Government work is classified as Public Domain in U.S. only)

Are any of the co-authors, employees or contractors of a National Government? Yes [✓ No]