MDCStream: A Stream Dataset Generator for Testing and Evaluating Stream Data Analysis Algorithms

Master Thesis

for obtaining the academic degree

Master of Science

as part of the study

Electrical Engineering and Information Technology

carried out by

Denis Ojdanić

student number: 01226704

Institute of Telecommunications

at TU Wien

Supervision:

Senior Scientist Dipl.-Ing. Dr.techn. Félix Iglesias

Univ. Prof. Dipl.-Ing. Dr.-Ing. Tanja Zseby

Erklärung zur Verfassung der Arbeit

Hiermit erkläre ich, dass die vorliegende Arbeit gemäß dem Code of Conduct – Regeln zur Sicherung guter wissenschaftlicher Praxis (in der aktuellen Fassung des jeweiligen Mitteilungsblattes der TU Wien), insbesondere ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel, angefertigt wurde. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In– noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Vienna, November 2019

Author's signature

Statement on Academic Integrity [1]

I hereby declare that this thesis is in accordance with the Code of Conduct rules for good scientific practice (in the current version of the respective newsletter of the TU Wien). In particular it was made without the unauthorized assistance of third parties and without the use of other than the specified aids. Data and concepts directly or indirectly acquired from other sources are marked with the source. The work has not been submitted in the same or in a similar form to any other academic institutions.

[1] Translation of the text above. The German version is the legally binding text.

Abstract

The importance of stream data analysis and in particular outlier detection is constantly increasing in times where more and more data is generated. Machine learning and data mining have classically assumed that patterns discovered in datasets are stable and permanent, but real-life applications commonly show that internal data structures change and evolve. Another challenge is the fact that data comes in streams and needs to be analyzed on a timely basis. The range of application for stream classification algorithms is large spanning from IT security, industrial and financial applications to medical data analysis and many more. To develop and enhance new or existing algorithms it is crucial to have flexible environments to test, compare and evaluate such algorithms.

During this thesis, we developed MDCStream, a MATLAB tool to generate multidimensional stream datasets for testing stream data analysis algorithms. MDCStream focuses on implementing diverse types of nonstationarities. The tool is based on MDCGen, a highly flexible static generator capable to produce a broad variety of multi-dimensional data scenarios. We refined MDCGen, developed MDCStream, and used datasets generated with MDCStream to evaluate state-of-the-art stream outlier detection algorithms when facing different kinds of concept drift.

Experiments showed that algorithms performed similarly, being MCOD the most remarkable in terms of accuracy and runtime. The tested algorithms, which are all distance and sliding window-based, showed to be especially sensitive to cluster inter-distances due to limitations when evaluating density differences inside analysis windows.

MDCStream can strongly help the design and evaluation of future classification algorithms. Researchers and experts in many different scientific and technical fields can benefit from MDCStream using it to create datasets and special corner cases to thoroughly test their algorithms. MDCStream ensures that data follow specific design conditions, variations and geometries, which may not happen naturally either in benchmark datasets captured from real applications, therefore creating a perfect test and evaluation environment.

Zusammenfassung

Die Bedeutung der Datenstromanalyse, vor allem bei der Erkennung von Ausreißern - den sogenannten "Outliern" - nimmt in Zeiten, in denen stetig mehr Daten generiert werden, konstant zu. Der klassische Ansatz bei maschinellem Lernen und Data Mining geht davon aus, dass Muster, die in Datensätzen erkannt werden, stabil im Verlauf der Zeit bleiben und sich nicht verändern. Beispiele aus dem echten Leben zeigen jedoch häufig ein anderes Verhalten, wobei sich die interne Datenstruktur ändert und weiterentwickelt. Eine weitere Herausforderung besteht darin, dass Daten als Ströme mit hohen Datenraten empfangen und möglichst zeitnahe analysiert werden sollen. Die Anwendungsbereiche von Klassifizierungs-Algorithmen sind groß und umfassen unter anderem IT Sicherheit, industrielle und finanzielle Anwendungen, medizinische Datenanalyse und viele mehr. Um Algorithmen zu optimieren oder gar neue zu entwickeln, braucht es flexible Umgebungen, um diese testen und vergleichen zu können.
Im Verlauf dieser Masterarbeit entwickelten wir MDCStream, ein MATLAB Tool, welches multidimensionale Test-Datenströme generiert, um Algorithmen für die Datenstromanalyse zu evaluieren. Der Fokus von MDCStream liegt auf der Implementierung diverser Typen von Nicht-Stationaritäten. Das Tool basiert auf MDCGen, einem höchst flexiblen statischen Daten-Generator, der verschiedenste Varianten von multidimensionalen Testszenarien produzieren kann. Im Zuge der Arbeit optimierten wir MDCGen, entwickelten MDCStream und verwendeten die von MDCStream generierten Datensätze, um hochmoderne Outlier Detection Algorithmen für Datenströme auf verschiedene Typen von Concept Drift zu untersuchen.
Die Experimente zeigten, dass die Algorithmen ähnliche Resultate produzieren, wobei MCOD, sowohl in der Genauigkeit als auch bei der Laufzeit, heraussticht. Die getesteten Algorithmen, welche alle auf Distanzberechnungen und "Sliding Windows" beruhen, sind sensitiv auf die Distanzen zwischen den Clustern. Dieses Verhalten entsteht aufgrund von Limitierungen bei der Evaluierung unterschiedlicher Dichten innerhalb des Analyzefensters.
MDCStream kann bei dem Design und der Evaluierung von zukünftigen Klassifizierungs-Algorithmen eine immense Hilfe darstellen. Forscher und Experten in vielen verschiedenen wissenschaftlichen und technischen Gebieten können von MDCStream profitieren. Mit MDCStream werden Testdaten und Spezialfälle einfach generiert, um Algorithmen gründlich zu testen. MDCStream gewährleistet, dass generierte Daten spezifische Anforderungen, Variationen und Geometrien erfüllen, was nicht immer von Benchmark Datensätzen, die durch reale Anwendungen erstellt worden sind, behauptet werden kann. Dadurch stellt MDCStream eine perfekte Test- und Evaluierungsumgebung für Klassifizierungsalgorithmen dar.

Acknowledgments

I would first like to thank my thesis supervisor Félix Iglesias from the Institute of Telecommunications of the Technical University of Vienna who was my primary advisor during the course of the master thesis. His door was always open and he helped me with his expertise in the field of data analysis when I ran into trouble. Many thanks to Tanja Zseby for the great lectures during of my master studies which awakened my interest in the research conducted by her and her colleagues at the Institute of Telecommunications and for accepting me to do my master thesis under her supervision. Furthermore, I want to mention Alexander Hartl who assisted the project which benefited the progress of my thesis.

Many thanks to my parents Aida Ojdanić, Boris Ojdanić and my sister Andrea Ojdanić for supporting me through the course of my studies. Moreover, I want to express my gratitude to my girlfriend Sarina Haslinger for enduring my long hours of work and listening to me talking about thesis related problems.

Finally, I want to mention my work colleagues at Frequentis who taught me a lot about the correct approach to software development. Special thanks goes to Thomas Ederer, Nicole Fruehwirth, Christian Haas and Gerhard Kalab. For giving me additional tips for my master thesis I want to mention Maximilian Hantsch-Köller. They all contributed in some fashion, whether providing feedback or valuable lessons in software engineering, to help me finish my master thesis.

Contents

1 Introduction

1.1 Background

In the present time we witness unseen amounts of data being produced and collected. Nowadays, with the advances in technology almost any electronic device may be capable to generate and collect data. A paramount example of stream data environments is the Internet or, in a more general perspective, network communications.
Since there is such a variety of data sources, the analysis of large volumes of data becomes more and more challenging. Manifold machine learning algorithms exist trying to interpret and extract valuable information from data and more algorithms are in the process of being developed. Essential for the development and performance analysis of such solutions are proper testing environments such as real-life or synthetically generated data. However, due to numerous advantages such as control-ability and data validity, high quality dataset generators prove to be an indispensable tool for the development and analysis of classification algorithms.
A selected data analysis topic we will focus our evaluation on are outlier detection algorithms. Outliers are data points that differ significantly from the usually collected and observed data. A simple one dimensional example would be a sensor measuring the height of the water in a river. Usually, the river contains a balanced water level and the measurements are within certain boundaries. However, if there is a lot of rainfall, the water rises, perhaps a flood occurs and as a consequence we get measurements that differ distinctively from the norm and thus represent outliers.
Data analysis today is applied to much more complex scenarios than the one dimensional example given above. A very popular field is network security and intrusion detection. Monitoring IP packets received on a particular network offers a scenario where huge amounts of multidimensional data needs to be analyzed on a timely basis. The attack on a network resembles a small and distinct part of the data monitored though is of utmost importance to be detected. Such attacks usually manifest themselves as outliers in the datasets. Another example is video surveillance e.g. the monitoring of a runway on an airport using a video tracking algorithm. Any unusual data, such as animals or people on the runway, need to be detected immediately to guarantee a save landing or take off. Other fields of application are the financial sector, system health monitoring, detecting ecosystem disturbances and many more.
The analysis of evolving data using classification algorithms is a common task in many modern technology applications increasing the demand and importance for high quality data set generators.

1.2 Motivation

Numerous classification algorithms exist trying to interpret data in manifold ways and many scientists work on improving or developing new algorithms. Novel algorithms and methodologies need to be tested in a wide variety of scenarios before being used in real applications. There are two approaches to face this problem: $a)$ using dataset collections captured from real applications or $b)$ using synthetic data created by dataset generators. As thoroughly explained in Section 2.2 there exist a variety of dataset generators. Even the MOA framework itself offers some possibilities to generate own test data. However, most of these data generators are either devised for very specific fields, like the SASE generator [17], or they do not offer extensive control and configuration options to generate a multitude of diverse and distinguishable datasets.
Hence, we propose MDCStream, a tool to generate multidimensional, time-arranged data for testing stream data classification, clustering, and outlier detection. MDCStream is highly configurable and offers maximum control over the produced datasets. Additionally, to demonstrate the capabilities of MDCStream, we selected the topic of outlier detection to test and evaluate a set of relevant algorithms using our generated data sets.

1.3 Goals

The fundamental goals of this theses are:

- **Develop a stream dataset generator**: We developed MDCStream a flexible tool to create test datasets for stream data analysis. MDCStream is highly configurable enabling the user to create various scenarios for thorough testing of clustering or outlier detection algorithms. The focus of MDCStream lies on creating stream data test scenarios with different types of concept drift. For more information refer to Sections 3.1, 3.2 and 3.3.

- **Creating diverse test datasets**: We created a collection of various datasets implementing different types of concept drift and nonstationarity challenges in order to analyze and compare stream outlier detection algorithms. For more information refer to the Section 4.2.

- **Evaluation of outlier detection algorithms**: We used stream outlier detection algorithms provided by MOA, a popular open source framework for data stream mining and conducted tests with the datasets created by MDCStream. The algorithms we analyzed are SimpleCOD, MCOD, ApproxSTORM, ExactSTORM and AbstractC. The algorithms are explained

in more detail in Section 3.4. For the evaluation of the classification results produced by MOA's algorithms we utilized various metrics. An overview of the metrics is given in Section 3.5,

1.4 Methodology

To achieve our goals of developing a stream data generator and evaluating outlier detection algorithms we did the following:

- *Refinement of MDCGen:* MDCGen is an open-source tool to generate static multidimensional clustering test data sets. Our first step was to refine MDCGen by fixing bugs, refactoring the code, improving configuration options and increasing the overall user-friendliness.
- *Developing MDCStream:* We created MDCStream, which is a highly configurable tool, to generate multi-dimensional stream data sets with special focus on nonstationarity and concept drift. MDCStream builds on MDCGen and takes static data generated by MDCGen and transforms it to stream data by adding a time component to each instance.
- *MDCStream wrapper:* We developed a wrapper for MDCStream to hide the vast configuration possibilities behind an easy verbose configuration. The wrapper simplified the task of generating data sets, as it automatically generates multiple data sets using the verbose configuration as a basis. Additionally, the wrapper supports transforming data to a format which is necessary to run MOA's outlier detection algorithms.
- *Developing of MOA wrapper:* To run experiments, we implemented a wrapper using MOA's public API interface. The MOA wrapper enables us to easily configure the classification algorithms and to automatically conduct experiments using data generated with MDCStream.
- *Devising datasets:* We designed a set of twelve different test scenarios focusing on cluster movement, stationarity and inter cluster distances.
- *Automatic evaluation:* Finally, we developed scripts for automatic analysis of the classification results using a set of evaluation metrics proposed in literature.

1.5 Structure of the Thesis

A short overview of the thesis structure is given in the following list:

- Section 1 gives an overall introduction and briefly summarizes the contents of the thesis.
- Section 2 gives a general introduction to the topic of stream data analysis in Section 2.1. Then, an overview of state of the art dataset generators is given in Section 2.2. Lastly, an introduction to the topic of outlier detection is given in Section 2.3.
- Section 3 gives a detailed description of what was done during the thesis to achieve the above listed goals. First, all building blocks for MDCStream are explained in the Sections 3.1, 3.2 and 3.3. Then the MOA framework is described in Section 3.4 along with the configuration parameters for the outlier detection algorithms we used to conduct our experiments. Finally, the evaluation metrics we selected are explained in Section (3.5.
- In the Section 4 the experiments are described in more detail. First the datasets and their focus are presented in Section 4.2 and then the results produced by MOA's algorithms are discussed in Section 4.4.
- In Section 5 an overall conclusion for the thesis is given.
- In Section 6 the references are listed.
- Finally, in the appendix A some example Matlab scripts are provided.

2 State of the Art

Due to modern technologies and digitization more and more data gets generated and offers vast possibilities once it is analyzed and interpreted correctly. Data analysis is becoming an increasingly important research field and topics like stream data analysis, real-time analysis and big data are currently some of the main challenges. In this section, we want to give a brief overview of the state of the art of stream data analysis and the emerging topic of concept drift that is gaining importance in recent times. Then, we will explore state of the art synthetic dataset generators and real-life datasets in Section 2.2. Finally, we will focus our attention on outlier detection algorithms and discuss various approaches to solve this problem in Section 2.3.

2.1 Stream Data Analysis

2.1.1 Data streams

Definition of data streams:
A data stream is a potentially endless temporary ordered sequence of data points

$$..., p_{t-1}, p_t, p_{t+1}, ... \tag{1}$$

whereas p represents a data point, object or instance and t the time stamp. An object is a container holding various features. The number of features is the dimension of the object and describe the properties of it. Data streams can be characterized as a continuous flow of huge amounts of data points typically incoming at a very high rate. A crucial characteristic of data streams is their non-determinism or susceptibility to change. The overall distribution of incoming data points may vary over time making classification more complex.

2.1.2 Difficulties working with data streams

Classifying data that arrives in an ever-changing stream is not an easy task. The main reasons are [9]:

- *Transient objects:* Data points and their meaning concerning their current state are transient. As time passes, data points become out of date and need to be discarded. For the algorithm it is difficult to decide how long a data point is relevant.
- *Infinity of the stream:* Incoming data streams are possibly infinitely long. Many algorithms are developed that need the whole picture with all data

points before classifying the data. In stream data applications this prerequisite cannot be met. Algorithms need to construct their proper model of the data when not all data points have been analyzed. Using this model algorithms are able to classify further data points.

- *Arrival rate:* Determining the nature of a data point, whether it is an outlier or not, requires some computational effort and thus some time. Depending on the arrival rate of the incoming data, computers running the algorithms may be overloaded. Solutions using multiple workstations to increase computational power may help, however additional effort needs to be included in developing algorithms that support parallel computing.

- *Concept drift:* The overall image of the data may change over time. Old and novel classes may emerge and clusters can change their shapes, placement, and prominence, disappear slowly or abruptly, and also reappear. The perception of outlierness during different time periods is also versatile. Algorithms need to be able to assimilate the model of the data that they create to overcome issues with concept drift. In the next Section 2.1.3 concept drift is more thoroughly explained.

- *Uncertainty:* A lot of data is available and produced that can be analyzed. However, there is no guarantee that the data is complete and error-free. Data may be missing, inconsistent or erroneous.

- *Analysis cost:* Both instance-based and model-based classification methods face problems with analysis cost. Decisions and classifications need to be provided on a timely basis requiring algorithms to quickly evaluate a large amount of historical data.

2.1.3 Concept drift

During the task of stream data analysis there arises inevitably the problem that one can never know the complete picture since the incoming data stream might be infinitely long. A model has to be learned from a base dataset to be able to make future predictions. The task of the analysis algorithm is the mapping of the input data to a classified output value. Usually, this mapping is assumed to be static, meaning that incoming data is always to be interpreted the same way. However, in many cases this assumption is erroneous and the relationship between input data and output value changes over time.
Concept drift refers to the changeability of the input data in unforeseen ways and thus invoking the need to adapt the interpretation and mapping characteristics of the classification algorithm. A major problem in many real-world examples

is the so-called *hidden context*. Usually, there exists a hidden and indirect dependency that may influence the distribution of the input data. A typical example is weather prediction, where the season is not explicitly specified in the temperature data, but may influence it. Another example is vacation destinations over the years that are influenced by the safety statistics of countries as hidden context. The cause of change is hidden and not known a priory to the classification method making predictions much more complicated.

Figure 1: *Types of concept drift. Image based on [36]*

Through literature there are four main types of concept drift: *sudden*, *gradual*, *incremental* and *reoccuring* contexts as depicted in Figure 1. *Sudden* concept drift refers to an abrupt change in incoming data and this change remains. A real-life example is a student who finishes his studies and starts to work and suddenly his income increases abruptly. *Gradual* concept drift refers to changes in data that are not abrupt but with a phase where both the old and new concept coexist. An example would be the movement pattern of a student who starts working part-time during his studies. After a while, the student increases the number of hours in the company and finally starts working full time. *Incremental* drift is similar to gradual whereas the two patterns do not coexist. There is a smooth transition between the two phases. An industrial machine that slowly deteriorates in performance and quality shows an incremental drift. Lastly, *reoccuring* drift resembles contexts that alternate over time. Seasonal weather and temperature with colder temperatures in winter and warmer in summer represent a valid example. [41], [42]

2.1.4 How to deal with concept drift

There are various possibilities to deal with concept drift and a brief overview is given below [36] [37]:

- *Static model:* Establishing a static model a priory and assuming that the data stream is not changing. The static model may be used to detect concept drift as the accuracy of the predictions drops.
- *Periodic re-fit:* Periodically creating a new static model based on most recent historic data.
- *Periodic update:* Periodically updating the model using the static model as a basis.
- *Instance weighting:* Assigning a weight to data points depending e.g. on the time stamp to be able to establish models considering the importance and impact of data points. Older points have less ramification than more recent ones for example.
- *Adaptive ensembles:* Outputs of several models are compared and combined to form an ultimate decision. The adaptivity is achieved through model utilization, which rules from each model are considered for a decision at a certain point in time.
- *Situational awareness:* In some cases it may be possible for a system to identify the type of incoming data and select one out of several predefined models.

2.1.5 Data stream analysis use cases

In the following a list of examples for stream data analysis is given:

- A web server that collects information about the visitors, the IP address, date, time spent on the website and so on. Each visitor is a data point and each information collected about them represents a dimension of the dataset.
- Sensors in various applications like transportation, vehicles or industrial equipment collect and send data to a streaming application. The application monitors performance, system health and various other matters and invokes actions like informing personnel about failing equipment.
- Network intrusion detection systems monitor the network traffic in pursuit of unusual behavior.

- Telecommunication companies monitor huge amounts of phone call data which represent enormous sources of data that needs to be processed in real-time.
- The stock exchange is a field where data analysis in real-time may come in handy to be able to predict the market in the best possible manner.
- Solar power companies apply stream data analysis to monitor the panels, schedule service time and minimize the periods of low power production.
- Online gaming companies collect stream data about player-game interaction to enhance the gaming experience.
- In the field of health care, various applications are possible since patient monitoring produces a vast amount of data which, if processed on a timely basis, may produce alerts and save lives.

2.1.6 Stream data analysis vs time series analysis

Lastly, we want to mention the difference between stream data analysis and time series analysis. A time series is a set of numerical measures of the same entity taken at equally spaced intervals over time. The goal in time series analysis is to detect and establish a pattern on which future trends can be predicted. Stream data analysis, on the other hand, takes dataset features into account and characterizes data based on shapes drawn in the input space. Data streams are received over time and thus represent a time series in that sense. However, crucial information in a data stream is not necessarily encoded as a time series but is extracted by observing the correlation of dataset features. Furthermore, stream data analysis is done while data is incoming, whereas in time series analysis usually have the whole data set available for the evaluation.

2.2 Stream Data Generators

As described in the introduction, data stream generators are extremely important tools to help develop, test and evaluate classification algorithms. In the following, we will give an overview of some existing stream dataset generators.

- MOA (Massive Online Analysis) is an open-source framework developed in the University of Waikato, New Zealand, especially used for data mining in evolving data stream scenarios. Among the outlier detection algorithms that we use from MOA as described in Section 3.4, MOA also offers several

built-in data stream generators. *RandomRBF generator* is used to create datasets containing clusters. A fixed number of clusters is created whereas the center points get placed into the output space. Depending on weights datapoints are assigned to a cluster center point with a certain deviation. MOA offers additional functions like adding noise or creating concept drift by merging two different streams. Additionally, MOA contains other data stream generators to simulate various scenarios. A comprehensive list can be found in [15]

- *SASE* is an open-source system developed at the University of Massachusetts Amherst to perform pattern matching over data streams. For their evaluation process, they implemented a simple stream data generator mainly to simulate stock behavior. [17]

- *Ostinato* is a network traffic generator useful for network load testing and functional testing. Ostinato offers features to craft and send packets over various protocols, rates and streams.

- The AWS Kinesis Platform *KDG* (Kinesis Data Generator) generates data streams for evaluating Amazons streaming device service Kinesis Streams or Kinesis Firehouse. Kinesis collects log and event data from various sources such as servers, desktops, and mobile devices and the test data generator produces data containing such attributes. [19]

- Similar to Amazons KDG *faker.js* is a generator of massive amounts of realistic fake data in the browser and node.js. [20]

- Anand Narasimhamurthy and Ludmila I. Kuncheva proposed a framework to generate data simulating a changing environment in [21]. The algorithm accommodates STAGGER and Moving Hyperplane generation strategies and covers gradual changes, substitution, and systematic trends for data generation. Unfortunately, we could not find any implementation online to the data generator.

- Huawei introduces an open software called *streamDM* for mining big data streams. StreamDM includes various features and among them are some stream dataset generators. The Hyper Plan Generator generates classification problems based on rotating hyperplanes. The Random Tree Generator generates data streams by splitting features randomly and constructing decision trees. StreamDM offers a RandomRBF generator similar to the one implemented by MOA and finally a Random RBF Events generator which allows cluster movement and nonstationary behavior. [16]

- Patrick Lindstrom, Sarah Jane Delany, and Brian Mac Namee propose an approach to generate real-life stream data with concept drift in [22]. They use a 3-D driving game to produce data on how to drive around a track under various conditions. Classification algorithms learn the driving technique of the driver.

- Different collections of datasets exist online and may be used for testing and evaluating classification algorithms [23]. However, the quality and suitability for testing dedicated algorithms are often questionable. For example, testing outlier detection algorithms on labeled datasets may produce faulty results because the dataset uses wrong features and therefore labeled outliers are no geometric outliers. Lets consider a dataset examining documented cases of people having heart disease. Selecting features that correlate and affect whether a patient has heart disease, e.g. age, diet, alcohol abuse and so on would create a picture of the data where outliers are visible geometrically, meaning a person who is young, has a healthy diet and still suffers from heart disease would easily be spotted when the dataset is plotted. However, the problem is that usually improper features are added and outliers are no longer geometric outliers. Using the example of heart disease, unsuitable features would be the patients eye color or the patients height. Moreover, it may be difficult to find suitable datasets to thoroughly stress test a proposed classification algorithm. Here a tool would be more advantageous to device special corner cases to test the limits of the algorithm.

There exist various dataset generators and datasets online. However, mostly those generators are implemented for specific studies and not for general-purpose testing. Most of the above-stated tools lack options to control the features of the generated dataset like the number of dimensions, outliers or clusters, the ability to add concept drift, control cluster shape and overlap and so on. Likewise, the suitability for evaluating classification algorithms using exclusively datasets found online is questionable. Datasets might be faulty and not extensive enough to thoroughly analyze a classification algorithm.

2.3 Outlier Detection

Outlier detection is one of the most important research problems in data analysis. It aims to find objects that are exceptional, dissimilar and inconsistent with the rest of the dataset. A popular definition for outlier is given by Hawkins: *"An outlier is a data object that deviates significantly from the rest of the objects as if it were generated by a different mechanism"* [7].

Outliers may appear in datasets due to various reasons, however, because of their nature, they are very important to detect. Typically outliers will translate to some kind of problem and applications range from intrusion detection, fraud detection, system health monitoring, unauthorized access in computer networks, activity monitoring, video surveillance, motion detection, medical condition monitoring, and many more [8]. Furthermore, there are many different classes of outliers depending on the context, e.g. local, global, contextual, collective, extreme values, subspace to state a few. [9] [10].
First, we will first discuss requirements for a classification algorithm in Section 2.3.1 and give an overview of different solution strategies in Sections 2.3.2 and 2.3.3. Then, we will briefly discuss evaluation measures in Section 2.3.4 and focus our attention to types of outlier detection algorithms in Section 2.3.5. Finally, we will give an overview of MOA's stream outlier detection algorithms in Section 2.3.6.

2.3.1 Stream classification requirements

Before discussing some approaches to solve the task of classifying data streams, we establish a set of requirements that classification algorithms need to fulfill according to [43]

- Algorithms shall process one instance at a time
- Algorithms shall use a limited amount of memory
- Algorithms shall be capable to provide results and predictions at any time
- Algorithms shall be capable to react to concept drift and nonstationary scenarios

2.3.2 Block vs online processing

Typically there are two approaches to deal with data streams: block and online processing. Block processing processes groups of datapoints at a time as seen in Figure 2. Online processing, on the other hand, updates the model and the classification result after each data point see Figure 3.
Block processing is usually more memory efficient and online analysis allows much faster reactions to change in data. Both methods have their advantages and disadvantages, however, only the online processing fulfills the requirements stated above and is suitable for analyzing data streams. [44]

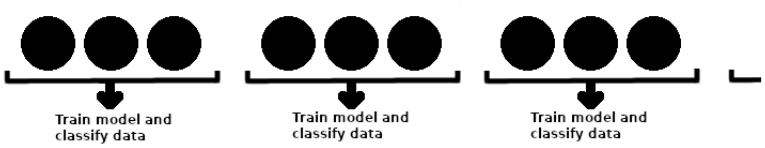

Figure 2: *Block processing. Image based on [44]*

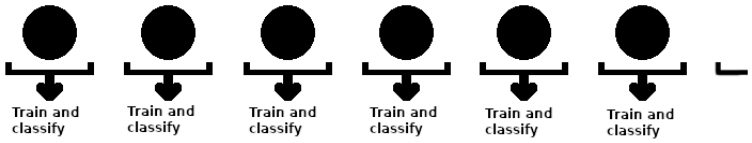

Figure 3: *Online processing. Image based on [44]*

2.3.3 Data management

To manage memory consumption there are some approaches to reduce the amount of data needed to establish a model. Additionally, for the analysis of data streams typically older objects are less important than recent ones and should contribute less to the evaluation. Most applications are interested in the most recent behavior. Therefore, various models have been introduced to set relevant data apart from outdated data. One approach is to decide based on each data point whether it should be considered for the learning model or not. Depending on the rules established, the learning model may be constructed e.g. based on different importance of features. Data points with high score in some features may be preferred to create a learning model. Another possibility may consider the time stamp of the data points to construct the model. However, different properties need to be implemented to decide whether a data point should be considered or not and important data might be missed due to misconfiguration.

Probably the most common approach, which considers all data points, uses different types of windows. Data within a window is in some form relevant to the classification algorithm. A window is a sequence of data points that are located within the window boundaries which may be timestamps or simply a certain number of most recent instances. The *landmark window* fixes different points in the data stream and the analysis is only performed from the current data point to the last landmark. In Figure 4 an illustrative example is given. The green lines mark the landmarks and the rectangles are data points. The blue rectangles mark data points that are considered for the current analysis. In the upper sequence, the algorithm performs the analysis for the three blue rectangles behind

the last landmark. The red rectangles mark data points that have not arrived yet. As some time passes and the two red rectangles have arrived, the algorithm performs another analysis as seen in the lower sequence in Figure 4. Only one rectangle is considered for the analysis since it is the only data point behind the last landmark.

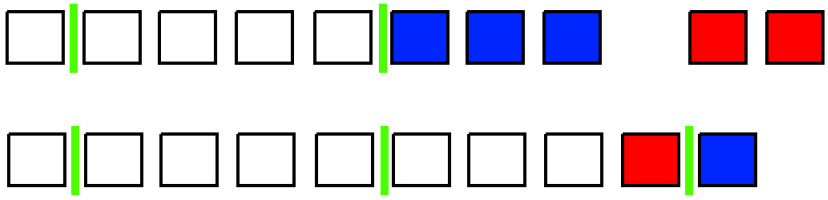

Figure 4: *Example of landmark window*

In contrast to that the *sliding window* has a fixed size, either a time interval or several data points and works on the principle of FIFO (first in first out). As new data points arrive, the oldest ones are discarded as seen in Figure 5. The window contains six data points and as the red rectangle arrives as a new point, the oldest one gets discarded. To enhance the impact of the most recent instances *decay functions* can be applied to the data inside the window. Most algorithms use the step function as a decay function, whereas all instances located within the window are of equal weight and the instances outside the window are ignored. [34]

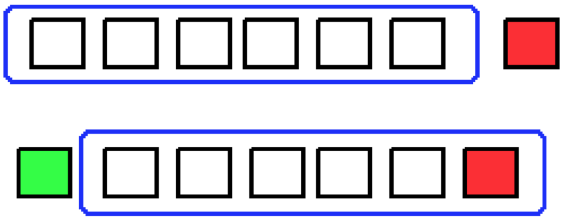

Figure 5: *Example of sliding window*

2.3.4 Evaluation

There are some key measures to evaluate the quality of a classification algorithm, namely:

- processing time,

- memory usage,
- prediction performance and
- capability to adjust to change in incoming data

The time to process an instance and the memory needed to do so should remain constant throughout the run time. Ideally, processing time and memory usage should be minimized as much as possible to allow real-time usage of the algorithm. However, those are simple measures to evaluate the performance of a classification algorithm. Metrics used to evaluate the predictive performance of classification algorithms are a bit more complicated to choose and evaluate. A study on evaluation metrics for stream data algorithms can be found in [39]. Lastly, the ability to adapt to changes may be monitored by comparing the drift reaction times of the algorithms. This is done by measuring the time between the first occurrence of the drift to the change in the model. More elaborate methods are discussed here [45]

2.3.5 Types of outlier detection techniques

Diverse techniques have been developed in the past years to detect outliers and in this Section we are going to give a brief overview.

Statistical methods

Statistical outlier detection methods use probability or distribution models to classify data [27]. Outliers are usually points with low probabilities. An example would be a Gaussian distributed dataset. The standard deviation can be used as a cut-off to identify outliers. Setting the cutoff to three standard deviations away from the median encapsulates 99.7% of data, meaning that three outliers would be expected in a dataset of 1000 datapoints that perfectly matched a Gaussian distribution.
Statistical outlier detection distinguishes between two kinds of methods:

- *parametric methods:* assume the distribution mode a priory. The statistical model can be build based on test datasets. In a stream data scenario, this method is not reliable, since the underlying distribution of the data may change over time.
- *non-parametric methods:* No assumptions are done and the model of the data is learned from the input data. This method produces good results when applied to data streams, although not for high dimensional datasets. [9]

Diverse statistical methods exist including approaches using distribution fitting [28], plain statistics [29], regression techniques [30], histograms [31] and many more.

Distance based methods

Distance-based methods detect outliers based on simple distance calculations between the data points. Most commonly the distance is calculated to the k nearest neighbors and outliers are data points with less than k neighbors within a radius of R. [11]. In Figure 6 the black point in the middle of the circle is considered an inlier if there are more than k neighbors within the circle indicated as red points.

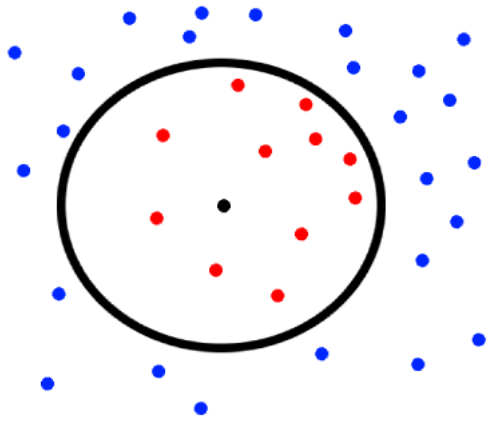

Figure 6: *K-nearest neighbors*

These methods offer good outlier detection and they do not need a priory knowledge of the underlying dataset and are also applicable to data streams. For data streams an additional parameter is introduced, the window size w. An outlier is detected if less than k neighbors are located within a radius of R within the current period w. [12].
However, these methods are not effective for high dimensional datasets as the volume increases and the available data becomes sparse and dissimilar. This phenomenon is referred to as the *curse of dimensionality*. [13].
A prominent example and one of the first distance-based outlier detection methods is DB(k,λ)-Outlier. [24]

Density based methods

Density-based outlier detection algorithms classify data points based on density values which are obtained by k-nearest-neighbor queries. Hence, density-based methods represent refined distance-based methods since densities are calculated based on distances. For normal data points, the density values obtained are similar while outliers produce significantly different results compared to its nearest neighbors.
A popular density-based algorithm is the local outlier factor (LOF) which was refined for data streams here [14]. Nowadays, many variants and improvements exist for the LOF algorithm. [25], [26]
Density-based methods are more effective in detecting outliers, however, they lack efficiency (are computationally expensive) and are more complicated than distance-based methods. [9]. Both density and distance-based methods furthermore rely strongly on the initial configuration.

Clustering based methods

Clustering-based methods first discover clusters in the dataset and thereafter extract outliers from the obtained result. Normal objects belong to clusters with a defined density and are located in the proximity of the cluster centroid. Outliers usually belong to sparse clusters far away from the determined cluster centroid. [32] [33]
Clustering methods are used to group similar objects and therefore proposed algorithms are not always suitable to detect outliers. Moreover, those algorithms heavily rely on the configuration and may produce misleading outlier detection results when analyzing datasets that do no fit cluster-like structures. [9]

2.3.6 Stream outlier detection algorithms by MOA

In the following an description of outlier detection algorithms provided by MOA is given:

SimpleCOD

SimpleCOD [40] is a distance and sliding window based algorithm for continuous outlier detection. The primary concern of this algorithm is to reduce the range queries and storage consumption and thus improve the overall performance. To achieve this, the framework takes advantage of the expiration time of objects which are known during runtime since there is a fixed window size. Using these predictions unnecessary computations and range queries are avoided.
SimpleCOD maintains two properties, a list containing all outliers and an event

queue, which schedules the necessary checks. The event queue holds all entries of inliers which are not safe and thus may still change to become outliers. Objects are put initially into this event queue with their corresponding expiration time, meaning they will be checked when they expire.

Object arrival: When a new object arrives a range query is performed. For all classified outliers that are found during this query, their respective succeeding neighbor count gets increased by one and subsequently they are checked whether they become inliers. In such a case that an outlier reaches the neighbor count of k, it is inserted to the event queue with a respective event queue processing time.
Furthermore, all inliers of the range query have their succeeding neighbor count increased by one. Finally, the incoming object is either classified as an outlier and is added to the outlier list, a safe inlier and is not added to any list or it is a normal inlier and it is put into the event queue.

Object departure: When an object departs or expires the most recent instances from the event queue are selected and checks are performed to see if their inlier status has changed. During the check their parameters are updated and they are classified as outliers or they remain inliers and are reinserted in the event queue.

To summarize, the event-based approach using the queue substantially reduces the range queries and checks required and thus makes the algorithm more efficient and applicable for stream data scenarios. Moreover, the event queue may be optimized by selecting suitable types of queues.

The special novelty of SimpleCOD is that it takes into account varying k parameters while R remains fixed. Since R does not change, the neighbors of each object remain the same and only the interpretation of whether an object is an inlier or outlier changes with the k parameter. The algorithm is the same as described above only multiple queries get performed with different k values. When an object expires the check if an object is an outlier is performed throughout all queries. Outliers are examined with a decreasing order of k and terminated as soon as a query is found where the object is an inlier. When a new object arrives again all queries are checked for a possible move of an outlier towards inlier set, again with decreasing order of k.

MCOD

Micro-cluster-based Continuous Outlier Detection MCOD is an algorithm that significantly reduces the amount of distance calculation for an arriving object.

Inliers are grouped to form small clusters and the range queries do not need to consider every data point but only cluster centroids. A micro cluster has to have a minimum size of $R/2$ and a minimum amount of $k+1$ elements. Generally, there are only $\frac{objects-in-window}{k+1}$ micro-clusters and an instance may have neighbors in various micro-clusters. The steps of the algorithm can be summarized as follows:

- Expired objects get discarded and the object counters of each microcluster are updated.
- For each new object the closest micro cluster and all microclusters within a range of $\frac{3}{2}R$ are detected.
- Depending on the distance of the closest microcluster there are two possible steps:
  - **closer than $\frac{R}{2}$:** The new object gets assigned to the closest cluster and the corresponding element count of the cluster gets increased. Additionally, the distance between the new object and all data points that are not contained in any cluster and have the microcluster of the new data point within its $\frac{3}{2}R$ distance is calculated. These points may change to become inliers or stay outliers.
  - **farther than $\frac{R}{2}$:** The object does not get assigned to a microcluster. A range query is performed between the new object and all objects that are not in microclusters and all objects that are within $\frac{3}{2}R$ proximity. If the number of neighbors exceeds a certain threshold a new microcluster is formed with the new object as the center. Otherwise, if none of the above is the case, an event queue queue-based algorithm is applied.
- If the size of the microcluster shrinks below $k+1$ the cluster gets dissolved and its objects are treated like the objects from the previous step.

Outliers are reported with the help of an event queue similar to SimpleCOD, however, the event queue does not contain elements that are within microclusters which significantly reduces the number of range queries.

ExactSTROM

The algorithm ExactSTORM consists basically of two procedures: the Stream and the Query Manager. The Stream Manager receives incoming data and updates the statistics and the Query Manager answers outlier queries.
ExactSTORM maintains the current window in an index structure called an ISB, the Indexed Stream Buffer. For more information about the ISB refer to section

4.4 of [4]. The ISB enables the query or the range query search to find neighbors more efficient. Each node of the ISB represents a data object. Two parameters are relevant for the algorithm, the counter $neighAfter$ and the list $neighBefore$, the former counting the succeeding data points within the radius of R and the latter containing a list of preceding nodes that are still not expired.
The basic operation of ExactSTORM is described as follows:

- An incoming data point is associated with a node and a range query search is performed in the radius R of the node. The search returns all preceding neighbors of the current node and adds those nodes to $neighBefore$. The current node acts simultaneously as a succeeding neighbor to all neighboring nodes already registered in the ISB and thus $neighAfter$ of each neighboring node gets incremented by one.

- Nodes that expire, are removed from the ISB. In other words, when the ISB is full and contains w number of elements, a new incoming data points is inserted while the oldest one is removed from the ISB.

- To determine if a data point is an inlier or outlier the sum of $neighAfter$ and not expired $neighBefore$ has to be greater or equal to k.

ApproxSTORM

ExactSTORM keeps the whole ISB in the memory. Large window sizes may exceed available memory resources making the algorithm not employable. ApproxSTORM implements two approximations to reduce the amount of data in the ISB and thus the amount of memory that needs to be allocated.

- **Reducing ISB size:** The nodes of the ISB can be categorized into inliers and outliers. Among inliers are furthermore the so-called safe inliers which will not belong to the group of outliers in any subsequent query. Safe inliers are defined when the $neighAfter$ count is larger than k.
  In order to reduce the size of the ISB randomly selected safe inliers are removed if the amount of safe inliers exceeds ρW. The random selection guarantees that safe inliers are distributed uniformly in the current window.
  To answer queries outliers and non-safe inliers, which during the course of the data stream might become outliers again, are maintained in the ISB.

- **Reducing node size:** Instead of storing a list of preceding neighbors for each node, ApproxSTORM only stores a fraction consisting of the ratio between the number of preceding neighbors of the current node in the ISB which are safe inliers and the total number of safe inliers currently in the

ISB.
At query time the number or preceding neighbors has to be estimated. This is done with the help of Equation 2, whereas the $ratio$ is the fraction described before, w is the window size, t is the query time and $nodeArrival$ the arrival time of the data point.

$$neighBefore = ratio + (W - t + nodeArrival) \tag{2}$$

To determine outliers the sum of $neighBefore$ and $neighAfter$ has to be greater or equal to k.

More information about ApproxSTORM and ExactSTORM can be found in [4].

AbstractC

AbstractC works similar to ExactSTORM only that it does not keep the exact indices of the preceding nodes, but a count of neighbors. Every data point that is received has by default an expiration date which depends on the window size. With this expiration date, it is possible to keep track of the exact neighbor count for each data point without keeping lists of node indices and thus reducing the amount of needed memory and range query searches.
Lets take a look at an example: Node A will expire within three increments and its lifetime neighbor count look is $lt\_cnt = (Inc1 : 3/Inc2 : 2/Inc3 : 0)$. Node A has currently 3 neighbors as stated at $Inc1$. On the next increment, one of node A's neighbors will expire and thus node A will have only two neighbors and so on. With the lifetime neighbor count, we have an exact overview of whether a node is an outlier or not. Upon inserting a new node B that is a neighbor of node A, the lifetime neighbor count of node A gets increased by 1 $lt\_cnt = (Inc1 : 4/Inc2 : 3/Inc3 : 1)$. Node B has its own lt_cnt which is constructed based on the expiration dates of its neighbors upon insertion.
More information about AbstractC can be found in [5].

3 Methodology

To perform thorough testing of classification algorithms allowing the developer to investigate various corner cases and configurations, we developed MDCStream, a tool capable to generate multidimensional stream datasets with special focus on concept drift and nonstationarity. MDCStream builds on MDCGen, which is a standalone tool to create static multidimensional clustering datasets and is described in Section 3.1. MDCStream takes the output produced by MDCGen and creates complex stream data test scenarios as described in Section 3.2. MDCGen and MDCStream are two highly configurable modules that produce multidimensional stream test datasets for classification algorithms. To hide the complexity of the vast configuration possibilities offered, we built a wrapper that encapsulates the two modules and offers easy configuration, see Section 3.3.
To conduct our experiments we used stream outlier detection algorithms provided by the popular open-source framework MOA. An insight into how the algorithms work was given in Section 2.3.6. An overview about the MOA wrapper and the algorithm configuration parameters is given in Section 3.4. Finally, we chose different metrics to evaluate the performance of algorithms as described in Section 3.5.

3.1 Description of MDCGen

MDCGen or Multidimensional Dataset Generator for Clustering is a tool to generate multidimensional datasets to test and evaluate unsupervised classification algorithms. It allows high-flexibility in configuration to create datasets with clusters in different shapes using various underlying distributions. Some of the features MDCGen offers are high configurability of cluster shapes, cluster separation, overlap and compactness, cluster rotation, correlation and the addition of outliers and noisy features.
During the course of the thesis, we developed MDCGen v2.0 based on the initial version v1.0 [6]. The major goal was to increase user-friendliness and readability of the code since it is open source and anyone should be able to use and adapt it to their special needs. The first major improvement was to refactor the whole code according to the Matlab Style Guide as proposed by Richard Johnson [1]. Furthermore, the code in v2.0 is modularized into separate functions to enhance test-ability. The configuration of MDCGen has been improved to increase user-friendliness as thoroughly explained in Section 3.1.2. During the code overhaul, various bugs were detected and fixed, see Section 3.1.4. Finally, missing requirements were added and written according to ANSI/IEEE Guide to Software Requirements Std 830-1984 [2] in Section 3.1.1.
In this section we are going to discuss MDCGen v2.0 in more detail and the

improvements over v1.0 that were done during the course of the thesis.

3.1.1 MDCGen Requirements

The requirements for MDCGen were refactored according to ANSI/IEEE Guide to Software Requirements Std 830-1984 [2]. Furthermore, they are grouped based on different topics and are listed in the Tables from 1 to 5.

For clarification of vocabulary used a glossary is provided in the following:

- **The tool**: The tool refers to the MDCGen algorithm.
- **Datapoint**: Is a point in multidimensional space defined with its coordinates.
- **Dataset**: Is a group of data points.
- **Centroid**: Reference point for cluster placement in the dataset.
- **Cluster**: Is a group of points whose distribution in the space is defined by one or many distribution functions.
- **Distribution function**: Is a statistical function that describes the possible likelihoods of data points being placed at a specified position.
- **Noisy dimension**: Dimension where data points are distributed uniformly on a large scale.
- **Outlier**: Point that does not belong to a cluster.
- **Offline adaptable**: Parameters for the program execution defined before the execution of the program.
- **Isometry**: Is a distance preserving transformation of the given data. (rotation)
- **Multivariate**: The distribution function is applied on data points independently for each dimension.
- **Radial based**: The distribution function describes dataset positions in regard to centroids.
- **Space**: Space is a region or scope consisting of all dimensions that are configured.

- **Subspace**: Is a space with a selected number of dimensions less than the total amount of dimensions configured.

Num	Title	Requirement
GEN-R001	Reproduce datasets	Output datasets shall be reproducible based on a random generator using a configurable seed.
GEN-R002	Dataset dimensions	The tool shall generate output datasets with 2 up to 200 dimensions. Note: More dimensions are possible depending on the hardware.
GEN-R003	Number of data points	The number of datapoints of the output dataset shall be offline adaptable.
GEN-R004	Number of outliers	The tool shall be capable to add an offline adaptable number of outliers to the dataset.

Table 1: *Requirements for the number of output datapoints*

Num	Title	Requirement
GEN-R005	Number of clusters	The number of clusters in the output dataset shall be offline adaptable.
GEN-R006	Cluster mass	The tool shall offer the possibility to specify an offline adaptable the number of datapoints per cluster.
GEN-R007	Cluster isometries	The tool shall offer functionality to generate isometries on selected clusters independently.
GEN-R008	Noisy dimensions	The tool shall offer capability to incorporate an offline adaptable number of noisy dimensions in the output datasets.
GEN-R009	Cluster correlation	The user shall be able to adapt offline correlation among cluster dimensions.

Table 2: *Requirements for cluster properties*

Num	Title	Requirement
GEN-R010	Cluster shape distribution	The tool shall offer the possibility to select between six default distribution functions that determine the shape of clusters.
GEN-R011	User defined distribution	The tool shall offer the possibility for user defined distribution functions that determine the shape of clusters.
GEN-R012	Mutivariate/Radial cluster shape	The user shall have the possibility to select offline between multivariate or radial based cluster shapes.
GEN-R013	Randomization for distributions	The tool shall be capable to randomly select distribution functions per cluster.

Table 3: *Requirements for cluster shape*

Num	Title	Requirement
GEN-R014	Cluster overlap	The cluster overlap shall be offline adaptable.
GEN-R015	Cluster overlap in subspaces	The tool shall be capable to generate clusters that are independent in overall space, but overlapping in subspaces.
GEN-R016	Cluster noisy in overall space	The tool shall be capable to generate clusters in subspaces that appear noisy and non clustered in overall space.

Table 4: *Requirements for cluster overlap*

Num	Title	Requirement
GEN-R017	Output dataset	The tool shall generate an output dataset with datapoints represented by their coordinates.
GEN-R018	Labeled cluster datapoints	The output datapoints belonging to clusters shall be labeled according to cluster membership.
GEN-R019	Labeled outlier datapoints	The output datapoints considered as outliers shall be labeled as outliers.
GEN-R020	Validity check	The tool shall offer a validity check containing performance indices for the generated output dataset.
GEN-R021	Output file format	The tool shall save the output results in a .mat file format.

Table 5: *Requirements describing the output*

3.1.2 MDCGen Architecture

The principal structure of MDCGen is depicted in Figure 7. MDCGen takes a dedicated user configuration as an input and creates an output dataset with the desired properties. In the following paragraphs we will illustrate the steps necessary more comprehensively.

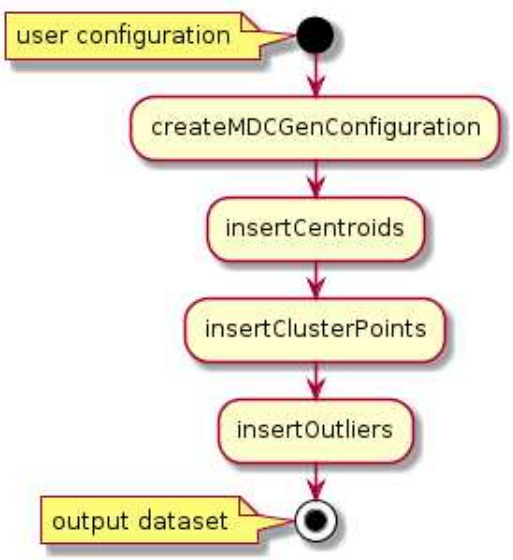

Figure 7: *MDCGen Architecture*

Create MDCGen Configuration

The first step of MDCGen deals with the configuration provided by the user. Consistency checks are performed and wrong input parameters generate errors and exit the program with dedicated error messages. In addition to the consistency checks, default values are assigned to configuration parameters not provided by the user. Once finished the user configuration is translated to a MDCGen configuration. This translation is necessary to enable an easier and more user-friendly interface to configure the desired dataset to be generated.

During this first phase of MDCGen, a grid is calculated to act as an anchor to place the clusters and outliers in later stages. Cluster and outlier placement is a crucial task for the generation of datasets and has a big impact on classifier performance. Distances between clusters need to be configurable and it must be assured that outliers are not placed on spots already taken by clusters. To solve this task a grid for every dimension is calculated based on Equation 3. Every

intersection of the grid acts as a possible spot to place a cluster or an outlier, except for the intersections at the very border of the space. Every dimension is divided by α_i equidistant hyperplanes.

$$\alpha_i = 2 + C_i * [1 + \frac{nClusters}{ln(nClusters)} + \frac{nOutliers}{ln(nOutliers)}] \tag{3}$$

α_i depends on both, the number of clusters $nClusters$ and the number of outliers $nOutliers$ to ensure enough grid intersections. C_i represents a configurable constant to enable the user a possibility of adjustment. Additionally, α_i may be configured directly by the user to control exactly the number of possible positions to place clusters or outliers.

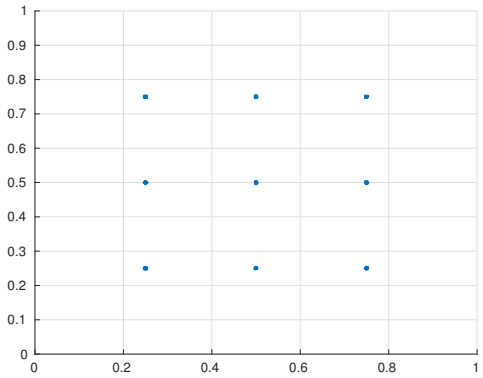

Figure 8: *MDCGen cluster centroids on grid intersections*

Insert Cluster Centroids

A cluster consists of a collection of data points that are close together concerning their euclidean distance. Clusters are placed in the output space on a grid intersection which calculation was described before. To position the clusters, anchor data points, the cluster centroids, are used, which represent the center of a cluster. These centroids are placed at random on a grid intersection as seen in Figure 8. In this example nine clusters are placed directly on the grid intersections. In the next step, the centroids are shifted by a random factor in the proximity of the grid intersection to smooth the cluster alignment as seen in Figure 9. An intersection may only contain one cluster centroid or one outlier, which in terms assures that there is no cluster overlap and no grouping of outliers.
In scenarios using multidimensional spaces with a large number of dimensions, an one-dimensional indexing to place clusters may become unfeasible. To avoid performance issues, Equation 4 is performed before the actual insertion of cluster

centroids. Based on the number of clusters a subset of base dimensions is selected containing enough intersections to place all centroids. Thus, in a subspace of the final output space, no cluster overlap is guaranteed and the remaining dimensions are filled at random.

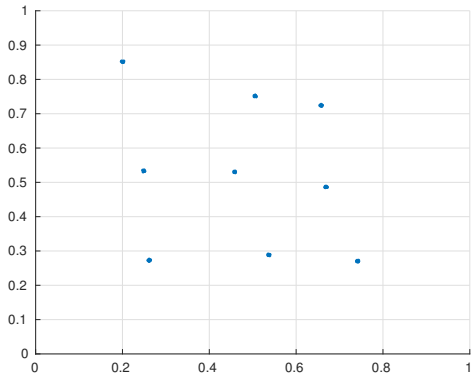

Figure 9: *MDCGen cluster centroids deviated from grid*

$$baseDimensions = 2 * nClusters + \frac{nOutliers}{nClusters} \tag{4}$$

Insert Cluster Points

After determining where the clusters ought to be placed, the next step is to create the actual point cloud and put it on the desired location in the output space. MDCGen offers various configuration possibilities for the shape of the final cluster point cloud. The user may select one of six distribution functions or define an individual distribution function. The six supported functions are:

- Uniform,
- Normal,
- Logistic,
- Triangular,
- Gamma and
- Ring shaped.

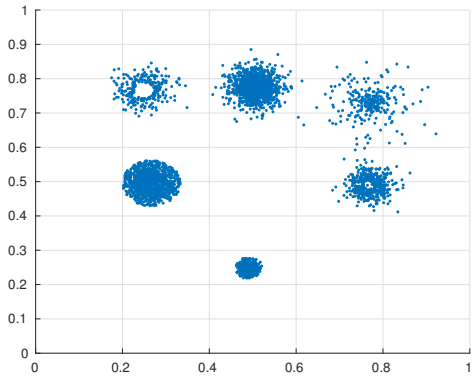

Figure 10: *Various cluster shapes defined by distribution functions*

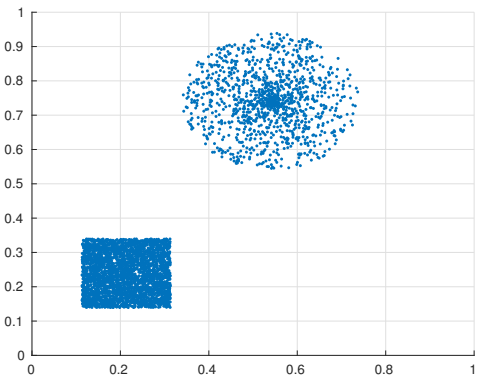

Figure 11: *Multivariate vs radial cluster shape*

These distribution functions define the allocation of data points in the point cloud as seen in Figure 10. In addition to those functions, the cluster shape may be multivariate or radial based. **Multivariate** meaning a distribution function is applied to each dimension of the cluster independently.
A **radial-based** configuration allocates the data points based on the selected distribution in the vicinity of the cluster centroid. MDCGen uses the following steps to create a radial based distribution:

- Generate Gaussian distributed points per dimension.
- Normalize those points to the same distance to cluster centroid forming a n-sphere.
- Apply the selected distribution function to shift each data point.

An example to illustrate the output of a radial based distribution using a uniform distribution function for each dimension is shown in Figure 12. The expected outcome is a circular cluster with a cone-shaped density distribution of data points. In the center of the cluster the density is large decreasing constantly as the radius increases.

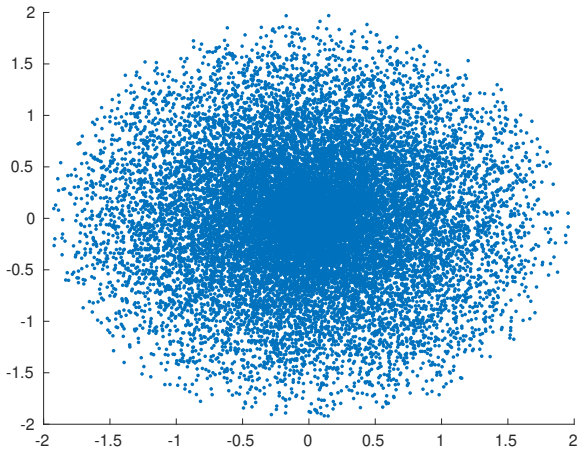

Figure 12: *Radial based uniform cluster from MDCGen v2.0*

Further steps are done to transform the data during this phase depending on the user configuration. The cluster point clouds might be rotated, a correlation factor might be applied or noise added to certain dimensions. [6]
Finally, the cluster point clouds are inserted into the output space by a simple vector addition between the cluster point cloud and the cluster centroid that was already placed in the output space.

Insert Outliers

The insertion of outliers works analogous to the insertion of cluster centroids. Free grid intersections are selected at random and the outliers are placed and shifted by a random amount.

3.1.3 MDCGen Configuration

MDCGen offers high-flexibility to configure the desired output dataset. In this section we are discussing the features offered by MDCGen.

Input configuration parameters

The configuration of MDCGen v2.0 was adapted to increase the user-friendliness. A list of all parameters is provided below.

- **seed**: [scalar] The seed for the random number generator to allow dataset reproducibility.
- **nDimensions**: [scalar] The number of dimensions in the output dataset.
- **nDatapoints**: [scalar] The number of cluster data points in the output dataset.
- **nOutliers**: [scalar] The number of outliers to be added to the output dataset. The total amount of output datapoints is the sum of nDatapoints and nOutliers.
- **nClusters**: [scalar] The number of clusters in the output dataset
- **clusterMass**: [array] The number of data points per cluster. The length of the clusterMass vector has to be equal to nClusters.
- **minimumClusterMass**: [scalar] The minimum amount of points that will be assigned to a cluster.
- **alphaFactor**: [scalar, array] alphaFactor is the C_i in Equation 3. If provided as a scalar alphaFactor is applied to all dimensions and as an array for each dimension independently.
- **alpha**: [scalar, array] This property sets α from Equation 3 directly. As a scalar it is applied to all dimensions and as an array for each dimension independently. If both alphaFactor and alpha are configured, alpha overwrites the alphaFactor.
- **scale**: [scalar, array] Scales the cluster compactness. As a scalar scale is applied to all clusters and an array for each cluster independently.
- **distribution**: [scalar, array, matrix] This property configures the distribution used to generate cluster data points. There are six default distributions available (1) Uniform; (2) Gaussian; (3) Logistic; (4) Triangular; (5) Gamma; (6) Ring Shaped and additionally the user may provide own distributions.
  Configured as a scalar the distribution is used for all clusters and dimensions. Configured as an array the distribution will be used per cluster and

if provided as a matrix each cluster dimension may have an individual distribution. If this property is set to zero then MDCGen randomly chooses a distribution function.

- **distributionFlag**: [array] It is a flag to enable or disable distributions. The length of the array has to be the same size as distributions are available. Without any user distributions, the length is six. (1) enables a distribution and (0) disables a distribution.

- **multivariate**: [scalar, array] Setting multivariate to (-1) the distribution will be applied to clusters radial based. Setting multivariate to (1) the distribution will be applied to clusters in a multivariate way. (0) randomizes this decision.
  Configured as a scalar the value is valid for all clusters and as an array, each cluster may have an individual configuration.

- **correlation**: [scalar, array] Sets the correlation. Configured as a scalar correlation is applied to all cluster dimensions and as an array for each cluster dimension independently.

- **compactness**: [scalar, array] This value determines the variance component for the distribution functions. Configured as scalar compactness is applied to all cluster dimensions and as an array for each cluster dimension independently.

- **rotation**: [scalar, array] Is a flag to enable or disable a rotation by a random amount. Configured as a scalar all clusters are rotated and as an array, only clusters with the rotation flag enabled are rotated.

- **nNoise**: [scalar, array, matrix] This property adds noise to the dataset. If configured as a scalar is determines how many dimensions will be noisy. If configured as an array the values mark which dimension of the dataset will be replaced by noise and if provided as a matrix the dimensions of each cluster may be replaced by noise independently.

- **validity**: [scalar] It is a flag to enable the validity check

  Silhouette: [scalar] Enable Silhouette validity check

  Gindices: [scalar] Enable Gindices validity check

- **userDistribution**: Additional user distributions may be defined. UserDistribution is a struct with two elements:

  binProbability: [array] The probability that values lie within a certain bin. The sum of these probabilities has to be equal to 1.

edges: [array] The edges of the bins in a range from -1 to 1. The length of the edges vector needs to exceed the length of the binProbability by one.

MDCGen Output

The MDCGen output is a Matlab struct object containing the following elements:

- **dataPoints**: [matrix] It contains all generated datapoints. Each row represents a datapoint and each column a dimension.
- **label**: [array] It contains the label for each datapoint. The label is either zero if the datapoint is an outlier or a positive nonzero number indicating that the datapoint belongs to a cluster. Clusters are randomly numbered during creation and data points belonging to a cluster are given the corresponding cluster number.
- **perf**: A struct containing the performance information.

3.1.4 Improvements done during Thesis

In this section we are going to list all the improvements and bug fixes done from MDCGen v1.0 and their respective solution for v2.0.

Radial based distribution bug

MDCGen v1.0 generates radially based distributed clusters using the following steps:

1. Generate uniformly distributed points per dimension.
2. Normalize those points to the same distance to cluster centroid forming a n-sphere.
3. Apply the selected distribution function.

Selecting, for example, a uniform distribution the expected outcome should be a cone-shaped cluster meaning many points in the middle and a decreasing sparsity towards the cluster borders. V1.0 does not achieve this requirement as clearly visible in Figure 13.
Using uniform distributions to generate points per dimension as proposed in step one does not produce a n-sphere with points uniformly distributed along the surface in step 2. At about 45 degrees there are more points located than at 0 or 90 degree and therefore a cross-shaped distribution emerges as seen in

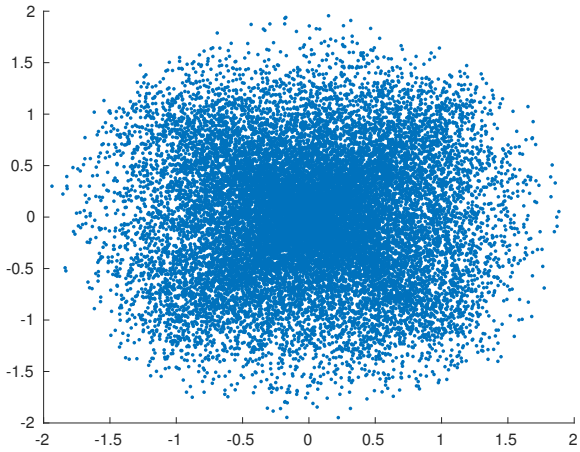

Figure 13: *Radial based uniform cluster from MDCGen v1.0*

Figure 13. To solve this issue the generation process had to be adapted slightly as proposed by [3] using a Gaussian normal distribution to generate the points in step 1 instead of a uniform distribution. Using this approach the points on n-sphere produced in step 2 are uniformly distributed along its surface resulting in a beautiful cone-shaped cluster as seen in Figure 14.

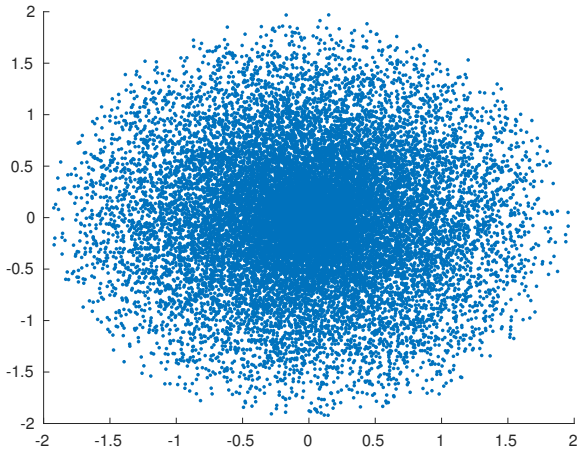

Figure 14: *Radial based uniform cluster from MDCGen v2.0*

Number of Intersections bug

1. Outliers are grouping to form unwanted clusters:

In MDCGen v1.0 artificial clusters could be formed by configuring many outliers as seen in Figure 15.

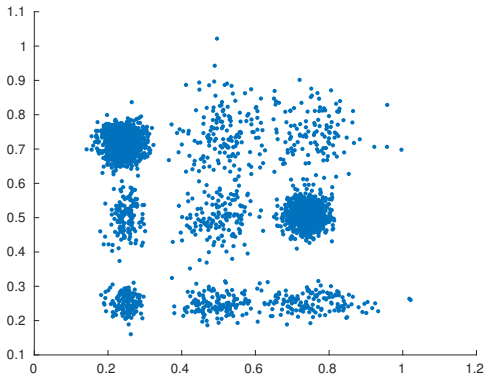

Figure 15: *Outliers forming artificial clusters*

To reproduce the problem using MDCGen v1.0 the following configuration should be used:

```
p.M = 2000; // in v2.0 nDatapoints
p.k = 2;    // in v2.0 nClusters
p.out = 1000; // in v2.0 nOutliers
```

In MDCGen v1.0 for **nIntersections** the name **cmax** is used. Cmax was calculated using the Equation 5 from [6] whereas **k** represents **nClusters**. The flaw however is that this equation only depends on the number of clusters.

$$\alpha_i = 2 + C_i * [1 + \frac{k}{ln(k)}] \tag{5}$$

A problem may arise if many outliers are configured. The resulting value for cmax or nIntersections would be too small to fit all outliers and "artificial" clusters may form from these outliers grouping around those few available intersections as seen in Figure 15.
To solve this problem Equation 5 was adapted to form Equation 3 as described in Paragraph *Insert Cluster Centroids* 3.1.2. The new equation has a dependency on both, number of clusters and the number of outliers. The output dataset

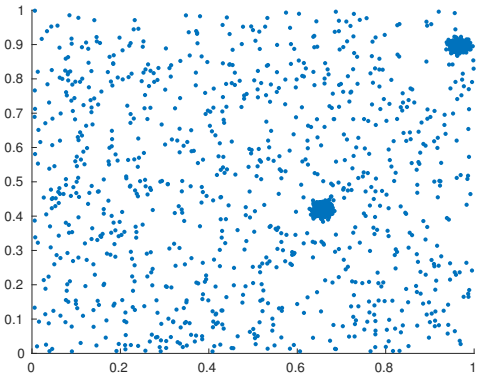

Figure 16: *Outliers do not form clusters*

produced with the adapted equation is displayed in Figure 16. Outliers are spread over the output space and do not form artificial clusters.

2. Problems with alphaN

The configuration options for **alphaN** were defined ambiguously in v1.0. Configuring alphaN as a positive number would set it to be the constant C in Equation 5. Configuring a negative alphaN would set the value for α in Equation 5 directly. Two different properties are mixed in one configuration value.
To solve this issue there are two configuration parameters in v2.0.

- alphaFactor: representing the C constant from Equation 5.
- alpha: sets α directly from Equation 5.

Furthermore, the code calculating the number of intersections produced an error using the following configuration in MDCGen v1.0:

```
p.k = 1;          // in v2.0 nClusters
p.out = 1;          // in v2.0 nOutliers
p.alphaN = 1;       // in v2.0 alphaFactor
```

Setting the alphaN or the alphaFactor to 1 limits the number of intersections and thus possible spots for clusters and outliers. In MDCGen v1.0 there was no mechanism to resolve this issue. All these bugs were solved by adapting the formula and making changes to the code. The new code solving this bugs is located in the $calculateNIntersections.m$ file.

MinimumClusterMass bug

In MDCGen v1.0 is a bug regarding the **km** which was renamed to **minimumClusterMass** in MDCGen 2.0.

This property sets the minimum amount of data points allocated per cluster.
To reproduce the bug set:

```
p.M = 200;  // in v2.0 nDatapoints
p.k = 4; // in v2.0 nClusters
p.km = 50; // in v2.0 minimumClusterMass
```

In this example 4 clusters would each have 50 data points whereas exactly 200 data points are available to distribute. MDCGen v1.0 gets stuck in a endless loop using this configuration trying to redistribute the data points. The bug was fixed for v2.0 and the code is located in the file $setClusterMass.m$.

Rotation bug

According to the definition of configuration parameters in [6] rotation can be configured as a scalar to be applied to all clusters or as an array to be applied to each cluster individually. In MDCGen v1.0 the feature to configure rotations for clusters individually did not work at all and was fixed for v2.0.

Ambiguous configuration for nClusters

The parameter **k** from MDCGen v1.0 was defined ambiguously. Configuring **k** as a scalar defines how many clusters the dataset should contain and if defined as array each value represents the datapoints per cluster. The number of clusters was determined by the length of the array.
To increase user-friendliness for MDCGen v2.0 this ambiguity was split into the following parameters:

- **nClusters**: [scalar] The number of clusters.
- **clusterMass**: [array] The number of datapoints per cluster. The length of clusterMass has to match nClusters.

Correlation bug

MDCGen v1.0 throws an error when the correlation is configured. The bug arises due to poor naming of variables using only single letters. In the code **p** gets overwritten by a function calculating the correlation. However **p** is used as the struct

containing configuration. Therefore, the following code runs into an exception, because the configuration is overwritten during the program execution.
In MDCGen v2.0 this bug is fixed using proper naming according to Matlab Style Guidelines [1].

DistributionFlag bug

MDCGen v1.0 throws an error if the **dFlag** and **d** are set to zero. Setting d to zero means that the distribution can be selected randomly by the algorithm. However, by setting the dFlag to zero all available distributions are disabled and an exception occurs.
In the new version 2.0 **d** is renamed **distribution** and **dFlag** is **distributionFlag**. The distributionFlag is configured only as an array and sets the availability of distributions. The distributionFlag has to have the same size as the number of available distributions. This property is only used if a distribution value is set to randomization (0).

Noise Bug

nNoise can be configured as a scalar, array and a matrix. Configured as an array each value represents a dimension to be replaced by noise. In v1.0 there was no border check e.g. if 3 dimensions are configured for the dataset the noise could add more unwanted dimensions if configured so.
In v2.0 noise is defined as follows:

- scalar: Depending on the value existing dimensions are replaced with noise. The value has to be smaller than nDimensions.
- array: The values of the array explicitly state which dimension will be replaced with noise. The values have to be smaller than nDimensions.
- matrix: Same as an array but for each cluster independently.

Rotation overwritten by noise

Classified as a known issue for v2.0 rotation applied to clusters is overwritten if noise is configured. Noise is added to the data after rotation is applied and therefore noisy clusters are not rotated.

Adaptions

The renaming of the configuration variables increases the user-friendliness drastically. Moreover, proper error messages are implemented in v2.0 to help the

user understand possible configuration errors.

3.2 Description of MDCStream

In order to create a stream dataset to test outlier detection algorithms we created MDCStream as an extension to MDCGen. MDCStream takes the output data created by MDCGen and adds a stream data label to each data point. In the following sections, we will discuss the functionality of MDCStream in more detail.

3.2.1 MDCStream Requirements

The requirements for MDCStream are written according to ANSI/IEEE Guide to Software Requirements Std 830-1984 [2]. Requirements are grouped based on different topics and are listed in the Tables from 6 to 10.

For clarification of vocabulary used a glossary is provided:

- **The tool**: The tool refers to the MDCStream algorithm.
- **datapoint**: Is a point in multidimensional space defined with its coordinates.
- **dataset**: Is a group of data points with their respective cluster membership label.
- **cluster**: Is a group of points whose distribution in the space is defined by one or many distribution functions.
- **sampling time**: Time value for each data point representing the absolute point in time the data point occurs.
- **time between samples**: Time that passes between the occurrence of two samples.
- **distribution function**: Is a statistical function that describes the possible duration of time between samples.
- **stationary process**: The global drawing of clusters does not change over time.
- **nonstationary process**: The global drawing of clusters changes over time. Clusters might appear or disappear as time passes.

- **stream data label**: The label containing the sampling time of a data point.
- **stream dataset**: A dataset consisting of data points represented by their coordinates, labels for cluster membership and stream data labels.

Num	Title	Requirement
STR-R001	Stream data label	The tool shall add an integer to every datapoint of an input dataset to simulate a timestamp.
STR-R002	Constant time between samples	The tool shall be capable to add constant time between samples.
STR-R003	Distribution function	The tool shall offer the possibility to select between five default distribution functions that determine the amount of time to pass between samples of the dataset.
STR-R004	Distribution function per cluster	The tool shall offer the possibility to select between five default distribution functions that determine the amount of time to pass between samples of the same cluster.
STR-R005	Simultaneous time samples	The tool shall generate an offline adaptable percentage of samples having the same timestamp.
STR-R006	Maximum number of simultaneous time samples	The user shall be able to configure a maximum number of consecutive samples having the same timestamp.

Table 6: *Requirements describing the time samples*

Num	Title	Requirement
STR-R007	Reproduce datasets	Output datasets shall be reproducible based on a random generator using a configurable seed.
STR-R008	Stationary	The tool shall allow stationary processes. This means the global drawing does not change over time. Clusters are visible throughout the whole simulated time.
STR-R009	Nonstationary	The tool shall allow nonstatoinary processes. This means the global drawing changes over time. Clusters may appear, disappear or remain as the simulated time advances.
STR-R010	Number of datapoints	The tool shall allow configuring number of datapoints for the stream dataset.
STR-R011	Refill clusters	The user shall be able to configure the clusters used as datapoint source to refill the dataset when number of samples exceeds number of input datapoints.
STR-R012	Refill dataset	The user shall be able to configure all clusters as a datapoint source to refill the dataset when number of samples exceeds number of input datapoints.

Table 7: *Requirements for dataset properties*

Num	Title	Requirement
STR-R013	Cluster start time	The user shall be able to configure a start time for each cluster.
STR-R014	Start after another cluster finishes	The user shall be able to configure cluster to start sequentially. This means the cluster start time is set to be equal to the cluster end time of another cluster.

Table 8: *Requirements for cluster start time*

Num	Title	Requirement
STR-R015	Cluster movement	The tool shall be capable to change the coordinates of the input dataset to allow cluster movement over time.
STR-R016	Movement speed	The user shall be able to configure how fast the input clusters are moving over time.
STR-R017	Direction change	The tool shall allow changes in the direction of the cluster movement.

Table 9: *Requirements for cluster movement*

Num	Title	Requirement
STR-R018	Input dataset	The input dataset shall consist of • a matrix containing the data point coordinates • a vector containing the labels for each datapoint indicating cluster membership
STR-R019	Output dataset	The output dataset shall consist of: • a matrix containing the data point coordinates • a vector containing the labels for each datapoint indicating cluster membership • a vector containing the stream data label
STR-R020	Output file format	The tool shall save the output results in a .mat file format.

Table 10: *Requirements describing the input and output*

3.2.2 MDCStream Architecture

The principal structure of MDCStream is depicted in Figure 19. MDCStream takes as an input the user configuration and the output dataset generated by MDCGen and produces a stream dataset.

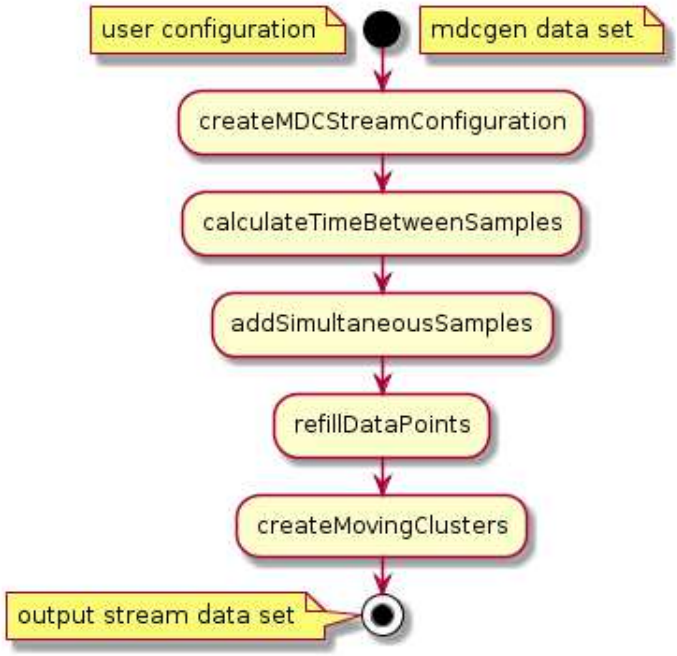

Figure 17: *MDCStream Architecture*

Create MDCStream Configuration

Similar to MDCGen, MDCStream takes a user configuration, validates the input parameters and outputs dedicated error messages to help the user correctly configuring the system. After this parameter check, MDCStream initializes parameters not configured by the user with default values and translates the user configuration into a MDCStream configuration.

Time Between Samples

The next step as per Figure 19 is to add time between samples to the data points thus creating the time label. MDCStream offers five distribution functions to create the time between samples:

- Uniform,

- Normal,
- Logistic,
- Triangular and
- Ring shaped.

In addition to the distribution function, MDCStream differs between two types of output datasets, namely stationary and nonstationary. Stationary time distributions create an output dataset where the global drawing of the data does not change over time. In other words considering the timestamp, clusters that are visible at the beginning of a dataset, are visible throughout the whole duration of the dataset. An example would be the network traffic observed on a node when the communication is stable. Regarding the time to live of the packets, observed time to live does not change as time passes. The distribution drawn by the time to live values remains stationary.
Nonstationary time distribution creates datasets where the global drawing does change over time. Thus, clusters might be appearing or dispersing in the course of the simulated time. Regarding the example given above, a network attack might trigger a nonstationary distribution, where suddenly a lot of traffic is received from another source address.
In Listing 1 the pseudocode for creating time between samples for the nonstationary case is summarized. First, the time between samples is calculated for each cluster independently. In the next step clusters are shifted in time, meaning that certain clusters will not appear until the simulated time of the dataset has reached a certain point. Another possibility for the nonstationary case is to shift the cluster times in a way that they appear sequentially.

```
FOR each cluster
  randomly mix cluster points
  calculate timeBetweenSamples based on distribution functions
END FOR

SET streamDataLabels as the cumulative sum of timeBetweenSamples

% shift time samples by a start time
SET ClustersToAdaptTimestamp TO first cluster
WHILE ClustersToAdaptTimestamp exists

  SET currentCluster To ClustersToAdaptTimestamp

  IF currentCluster has previousCluster
    SET previousClusterEndTime TO end time of previousCluster
  END

  ADD StartTime AND previousClusterEndTime TO streamDataLabels of
      currentCluster

  SET ClustersToAdaptTimestamp TO next cluster in queue

END WHILE

SORT all data points per streamDataLabels
```

Listing 1: Non stationary time between samples

Simultaneous Time Samples

Another feature of MDCStream is the possibility to configure a percentage of simultaneous timestamps to be able to test how classification algorithms cope with simultaneously incoming data points. To generate simultaneous instances MDCStream simply creates data points whose timestamps are identical. In the Listing 2 the code is summarized that creates simultaneous timestamps. Two configuration values determine the simultaneity, the percentage of simultaneous data points in the dataset and the number of consecutive simultaneous data points.

```
SET timeBetweenSamples of random dataPoints to ZERO
SET consecutiveSimultaneousCount to ZERO

WHILE NOT done

  SET done TO TRUE
  FOR each dataPoint

    IF timeBetweenSamples EQUALS ZERO
      INCREMENT consecutiveSimultaneousCount
      IF consecutiveCount BIGGER THAN maxConsecutiveSimultaneousAllowed
        SET done TO FALSE
        SWAP timeBetweenSample of current dataPoint with non zero
            timeBetweenSample
      END IF
    ELSE
      SET consecutiveSimultaneousCount TO ZERO
    END IF

  END FOR

END WHILE
```

Listing 2: Simultaneous time samples

Extend MDCGen data template

MDCStream allows extending the cardinality of the underlying MDCGen dataset if desired by the user. To achieve this, MDCStream uses the MDCGen dataset as a template and adds further data points while keeping the initial MDCGen configuration. The pseudocode for extending the cardinality is presented in Listing 3. New data points are created with the same features but with a later timestamp than the template data points. Therefore, when simulating the dataset, first the template data points are visible and afterward, the refilled points appear. The user may also configure only certain clusters to be refilled to extend the life of those clusters for example.

```
IF desiredNumber SMALLER THAN numberOfDataPoints
  RETURN first desiredNumber of dataPoints
ELSE
  IF refill whole dataset
    COPY dataPoints TO newDataPoints
    APPEND newDataPoints to the end of dataPoints from the dataSet
    ADD last StreamDataLabel TO newStreamDataLabels % shift stream data
        label to the end
    APPEND newStreamDataLabels to streamDataLabels of dataSet
  ELSE % refill selected clusters
    CREATE refillPoll from dataPoints of selected clusters
    CREATE new dataPoints and streamDataLabel from refillPool
    APPEND to dataSet and streamDataLabels
  END IF
  RETURN dataSet
END IF
```

Listing 3: Refill data points

Moving Clusters

An important feature of MDCStream is the cluster movement. The user may configure the statically placed clusters to move at a given speed and direction over time. This feature together with the nonstationarity simulates concept drift. In Listing 4 the pseudocode for cluster movement is summarized. To create cluster movement a displacement vector is added to the cluster data points. The cluster data points are segmented according to their timestamp. For example, if we have a cluster with 100 data points we can segment the data points into five groups of 20 points. The first twenty points remain unchanged and for the remaining points we add the displacement vector. Then the second segment of points remains unchanged and we add again the displacement vector to the remaining 60 points and so on. In the end, the first segment of data points was shifted zero times, the second was shifted by one displacement vector, the third by two displacement vectors and so on. A visual example of the functionality of the displacement vector is given in Figure 18.

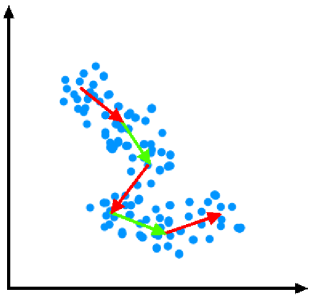

Figure 18: *Displacement vector*

```
FOR each cluster

  IF displacement AND displacementRate configured for current cluster
    calculate displacementVector
    SET ShiftBy to ZERO
    SET TimeForDisplacement

    FOR each data point of current cluster
      IF vectorChangeRate configured
        IF time to calculate new displacement vector
          calculate displacement vector
        END IF
      END IF

      IF TimeForDisplacement passed
        add displacement vector to current ShiftBy
        SET new value for TimeForDisplacement
      END IF

      ADD ShiftBy to current cluster point
    END FOR
  END IF
END FOR
```

Listing 4: Create moving clusters

3.2.3 MDCStream Configuration

A list of all parameters for the configuration of MDCStream v1.0 is provided below.

Input configuration parameters

- **seed**: [scalar] The seed for the random number generator to allow dataset reproducibility.
- **stationary**: [scalar] Stationary is a flag (0) meaning nonstationary and (1) meaning stationary.
- **tbsDistribution**: [scalar, array] This property configures the distribution used to generate stream data label. There are six default distributions available (1) Uniform; (2) Gaussian; (3) Logistic; (4) Triangular; (5) Gamma; (6) Ring Shaped. Specifying (0) the distribution is selected randomly. Configured as a scalar the distribution is used for all clusters. Configured as an array the distribution will be used per cluster if a nonstationary dataset is configured.
- **mu**: [scalar, array] Represents the mean of the distribution. As a scalar it is applied to all clusters and as an array it is being applied to each cluster individually.

- **sigma**: [scalar, array] Represents the variance of the distribution. As a scalar it is applied to all clusters and as an array it is being applied to each cluster individually.
  Mu and Sigma should be configured to generate positive values. If a distribution is placed close to zero negative values might be generated. In the implementation these values are simply multiplied by -1 meaning the distribution configured is not valid anymore. To obtain consistent data it is recommended to use a combination of mu and sigma that do not generate negative values.

- **simultaneous**: [scalar] The percentage of datapoints that should occur at the same time thus have an equal stream data label.

- **maxSimultaneous**: [scalar] The maximum number of time samples that occur at the same time.

- **startTime**: [scalar, array] Specifies a time offset for the beginning of a cluster. As a scalar it is applied to all clusters and as an array it is being applied to each cluster individually.

- **startAfterCluster**: [array] Specifies a cluster to start after another cluster has finished. By setting a value to zero a cluster does not start after another cluster.

- **nTimeSamples**: [scalar] number of points for the output dataset

- **refillClusters**: [scalar, array] This property is used if nTimeSamples is larger than the number of datapoints of the input dataset. It is a flag deciding which clusters are used to fill the gap between input dataset points and nTimeSamples. As a scalar it is applied to all clusters and as an array it is being applied to each cluster individually.

- **displacement**: [scalar, array] Displacement is a factor multiplied with a random normalized displacement vector to create moving clusters. The larger this value the farther the cluster will be displaced.

- **displacementRate**: [scalar, array] This values represent the percentage of data after which the displacement vector is applied to the remaining datapoints. Setting it to 0.1 would mean that after each 10% of datapoints the following datapoints are displaced by a displacement vector. As a scalar it is applied to all clusters and as an array it is being applied to each cluster individually.

- **vectorChangeRate**: [scalar, array] These values represent the percentage of data after which a new displacement vector is calculated thus altering the direction of the cluster movement. As a scalar it is applied to all clusters and as an array it is being applied to each cluster individually.

MDCStream Output

The MDCStream output is a struct containing the following elements:

- **dataPoints**: [matrix] It contains all generated datapoints. Each row represents a datapoint and each column a dimension.
- **label**: [array] It contains the label for each datapoint. The label is either zero if the datapoint is an outlier or a positive nonzero number indicating that the datapoint belongs to a cluster.
- **streamDataLabel**: [array] It contains the stream data label or timestamp for each datapoint.

3.3 Design of MDCStream-wrapper

MDCStream is a tool that generates multi-dimensional stream datasets with special focus on concept drift and nonstationarity that builds on MDCGen. MDCGen creates static multidimensional datasets containing clusters and outliers. Both modules are highly configurable due to the vast features they offer. To hide the complexity of the numerous configuration possibilities, we build a wrapper to encapsulate the two modules. In Figure 19 the overall design of the wrapper is visible.
The main advantage of the MDCStream-wrapper is that the user only has to provide a simple verbose configuration without needing to have a deep knowledge of the sub-modules encapsulated by it. This simple user configuration gets translated to a MDCGen and MDCStream configuration and handed over to the respective modules. The MDCStream-wrapper automatically saves the configuration and the output dataset to the desired folder. Moreover, the output dataset is saved in .mat and .arff format to allow running experiments with the MOA framework.

3.3.1 Configuration of MDCStream-wrapper

MDCStream offers the following configuration options whereas words in brackets surrounded with high commas are to be entered in the configuration file. The key words in high commas are translated to MDCGen or MDCStream configuration

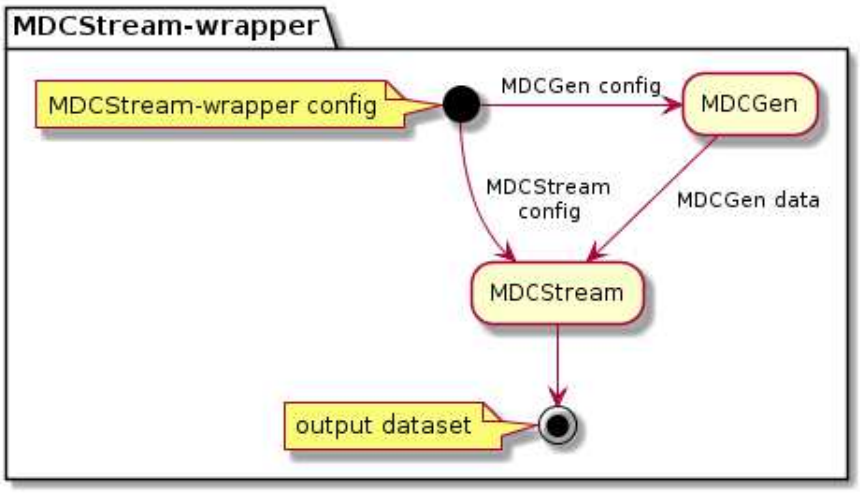

Figure 19: *MDCStream-wrapper Architecture*

values depending on default value ranges. E.g. *'many'* in the context of *dimensions* means the number of dimensions will be set to a random value between 3 and 100.
The output produced by MDCStream-wrapper is equal to the output of MDCStream and is described in Section 3.2.3.

- **scenarioName**: [string] The name of the respective scenario
- **nOfDataSets**: [scalar] The number of datasets to be generated from the configuration under the current scenario name
- **dimensions**: ['two', 'many'] Sets the number of dimensions.
- **stationary**: ['stationary', 'sequential', 'nonstationary'] Sets the stationarity of the dataset. Either the overall image does not change over time, the clusters appear in a sequential manner or the clusters appear and disappear in a random fashion.
- **outliers**: ['no', 'few', 'medium', 'many'] Sets the number of outliers.
- **clusters**: ['one', 'few', 'many'] Sets the number of clusters.
- **densityDiff**: ['no', 'few', 'many'] Defines the whether the densities in the dataset should be equal or differ. For instance the compactness, distribution, points per cluster or whether multivariate or radial based distributions are used.

- **space**: ['tight', 'extensive'] Specifies the characteristics of the output space. A tight space meaning the clusters are close together and an extensive meaning there is a lot of space between clusters.
- **movingClusters**: ['no', 'few', 'all'] Specifies whether clusters should be moving over time.
- **overlap**: ['no', 'yes'] Specifies whether clusters should be overlapping or not.
- **validity**: ['no', 'Sil', 'GOI', 'all'] This property activates the performance checks for the dataset.

3.4 The MOA Framework

For the evaluation and comparison of stream data analysis algorithms we chose the outlier detection algorithms implemented by MOA (Massive Online Analysis), a framework for data stream mining. MOA is a free open-source software project specifically developed for data stream mining and concept drift. MOA includes a set of algorithms for testing and evaluation. For our special needs, we created a MOA wrapper using the MOA API. The MOA wrapper lets us select and configure the outlier detection algorithms to run experiments using data created by MDCStream. The main advantage the MOA wrapper offers is full automation of running test scenarios and creating test results on an arbitrary amount of datasets.
The output obtained from the experiments is a simple list classifying each data point as inlier or outlier and adding an outlierness weight. Using this information combined with the metrics explained in Section 3.5 we can deduct our conclusions about the performance of the algorithms.

All MOA outlier detection algorithms share the same configuration parameters:

- **Window size w:** corresponds to the number of data points kept in memory for the classification of incoming data points.
- **Count threshold k:** The number of data points that have to be in the range of a new instance to be considered as an inlier. (The number of neighbors)
- **Distance threshold R:** The distance within which k neighbors have to be located to consider an instance as an inlier. (The neighborhood radius)

ApproxSTORM and ExactSTORM add two more configuration parameters: the query frequency f and the threshold to remove safe inliers ρ. An insight of which values were used for the experiments is given in Table 12.

3.5 Evaluation methods and metrics

In this section, we will discuss the metrics we used to evaluate the performance of the outlier detection algorithms. It is not part of the thesis to elaborate which metric is best to analyze stream data algorithms as this would go beyond the scope. We decided to choose common metrics as proposed in literature to evaluate our results.[39], [38].

To explain the evaluation metrics we will first elaborate some terms needed for better understanding. A classification system or algorithm returns results in four different categories. We will define the terms referring to our use case of outlier detection:

- **True positive:** Outliers correctly detected by the algorithm.
- **True negative:** Inliers correctly not detected as outliers by the algorithm.
- **False positive:** Inliers incorrectly detected as outliers by the algorithm.
- **False negative:** Outliers not detected as outliers by the algorithm.

True positive and true negative indicates that the prediction was correct, whereas false positive and negative indicate that the prediction was incorrect.

Precision

Precision is the percentage of true positives contained within all retrieved results. In other words, precision is the ratio between all correctly detected outliers and all data points that the algorithm classified as an outlier. Referring to Figure 20 this means dividing the inner dark blue rectangle - the true positives - by the inner rectangle - the true positives and true negatives.

$$precision = \frac{tp}{tp + fp} \tag{6}$$

Recall

Recall is the percentage of true positives retrieved from all relevant results. Following our example recall is the percentage of outliers that the algorithm detects regarding the total number of existing outliers in the dataset. Recall is the division of the inner blue rectangle - the true positives - by the whole blue section - true positives and false negatives - when referring to Figure 20.

$$recall = \frac{tp}{tp + fn} \tag{7}$$

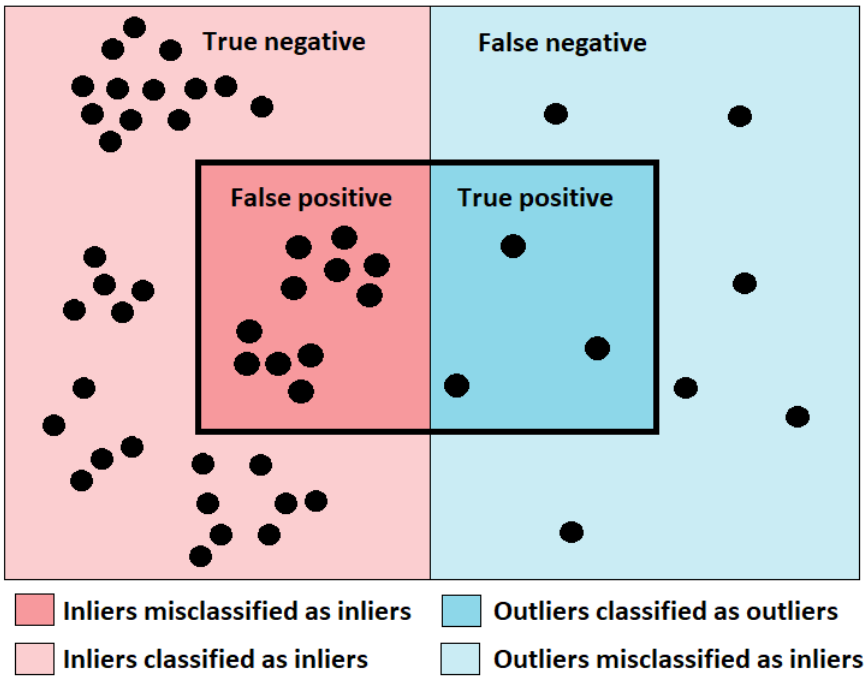

Figure 20: *Example for true and false positive and negative*

In the following sections we will discuss the metrics selected for our evaluation in more detail.

3.5.1 MaxF1

F1 measure can be interpreted as a weighted average of the precision and recall and the definition can be seen in Equation 8. F1 is the harmonic mean of precision and recall and ranges between 0 and 1. F score can produce better results than precision and recall alone.

$$F1 = 2 * \frac{precision * recall}{precision + recall} \tag{8}$$

3.5.2 Precision at n - P@n

Precision at n also denoted as $P@n$ represents the proportion of relevant or correct results from the total amount of results obtained. All data points get a ranking assigned and the relevant data points are the top n instances that have the highest ranking.

$$P@n = \frac{|relevant\_results|}{|n|} \tag{9}$$

An example for the use of $P@n$ would be a web search whereas the first page of the search results holds 10 entries. A $P@n$ with n set to 10 corresponds to the number of correctly found pages by the search engine that are displayed on the first page of the search results.
To obtain meaningful results it is very important to select a good value for n. Setting n too high would produce a deceptively low outcome even though all relevant results are found making the metric useless. For our case searching outliers setting n too high would suggest that even though all outliers are found in a dataset the $P@n$ value would be low. On the other hand, if n is too small or it the dataset contains too many outliers $P@n$ would produce very high values simply due to the fact that there are too few inliers to compare the result.
For our experiments, n has been set to the number of labeled outliers in the corresponding dataset.

3.5.3 Average precision - Ap

Usually when characterizing a classification algorithm precision and recall are observed simultaneously, since separately they would produce meaningless results. Rather than comparing the two values, it is more convenient to have a single number to evaluate the performance of an algorithm. The average precision combines precision and recall to obtain one performance value by adapting the threshold value n as seen in Equation 10. The average precision is basically the area under the curve drawn by precision and recall.

$$AP = \sum_{n=1}^{N} P@n * \Delta recall(n) \tag{10}$$

$P@n$ represents the precision at threshold n, N is the total number of data points and $\Delta recall(n)$ is the change in recall that happens between threshold n-1 and

n.

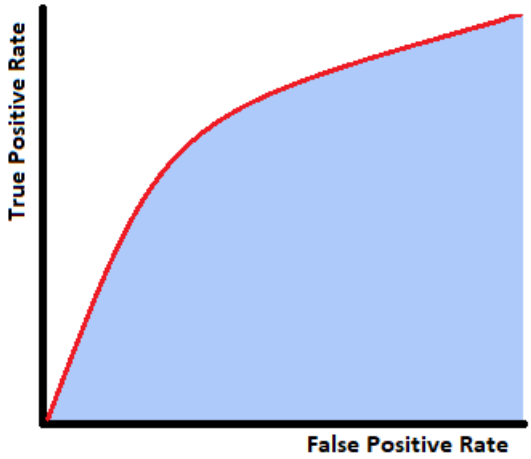

Figure 21: *ROC AUC curve. Image based on [46]*

3.5.4 RocAuc

The receiver operating characteristic curve is a two-dimensional plot that illustrates the performance of a classifier. The ROC curve is created by plotting the recall for all possible n values, see Equation 7, over the false positive rate, see Equation 11. In other words, it is a plot of correctly detected outliers versus the inliers ranked among the top n results over various thresholds n. The red line in Figure 21 shows the ROC curve.

$$FPR = \frac{fp}{tn + fp} \tag{11}$$

The ROC curve can be summarized by a single value namely the area under curve AUC which ranges between 0 and 1. The AUC value is an aggregate measure of the performance of the algorithm over all thresholds n. It is depicted by the light blue area under the ROC curve in Figure (21). An outlier detection algorithm that does not detect any outliers would have an AUC value of 0 and an algorithm that detects all outliers correctly would have a AUC value of 1.

3.5.5 Processing time

As described in Section 2.3.4 there are other evaluation measures besides the prediction performance to analyze how algorithms are performing. In addition to

the metrics specified in the previous sections we selected the processing time as another evaluation measure. We define the processing time for our experiments as the time span between the start of the algorithm and moment when the algorithm stops and gives us the final results.

4 Experiments

To illustrate the potential of MDCStream and the datasets it is capable of generating we run MOA's outlier detection algorithms over 240 of our generated datasets. In this section, we will first elaborate on what the major focus and differences are between our generated test datasets in Section 4.1. Then, we will give an overview of the configuration used for the MOA algorithms in Section 4.3 and finally, we present our results in Section 4.4.

4.1 Main focus of experiments

The datasets created for the experiments were grouped in three different types of problems, namely: different types of *stationarity*, different *input space size* relative to inter cluster distances and *cluster movement*.

4.1.1 Stationarity

Stationarity, together with moving clusters described in Section 4.1.3, simulate concept drift. Stationarity describes the overall image created by the data distribution and timestamps. For our experiments, we distinguish between three types of stationarities as described in the following paragraphs.

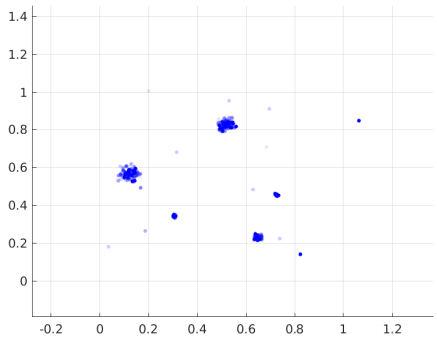

Figure 22: Stationary space t = 1000

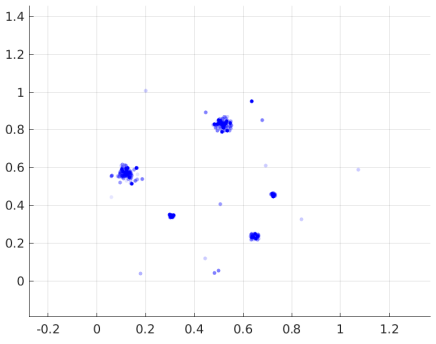

Figure 23: Stationary space t = 2000

Stationary

The overall image of the data distribution does not change over time. Clusters are visible throughout the whole simulated time of the dataset. In Figures 22 and

23 two excerpts are visible of a two dimensional dataset showing two different points in time. In both images five clusters are visible which remain stationary throughout the simulated time.

Nonstationary

In the case of nonstationarity the overall image of the dataset distribution changes over time. Clusters may appear or disappear at a certain point. From Figure 24 until 27 a dataset is displayed during various points in time. Figure 24 shows five clusters at the timestamp $t = 1003$. After a certain amount of time a sixth cluster $C(0.63, 0.6)$ appears as visible in Figure 25 and in Figures 26 and 27 clusters previously visible disappear.

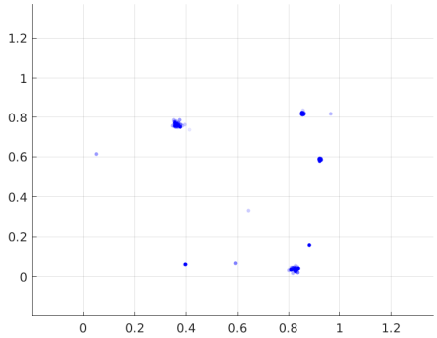

Figure 24: Nonstationary t=1003

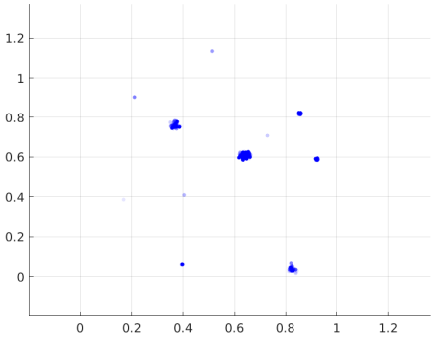

Figure 25: Nonstationary t=1578

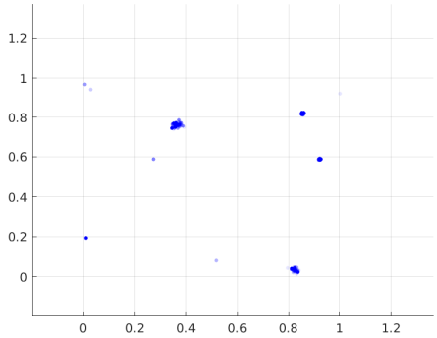

Figure 26: Nonstationary t=2688

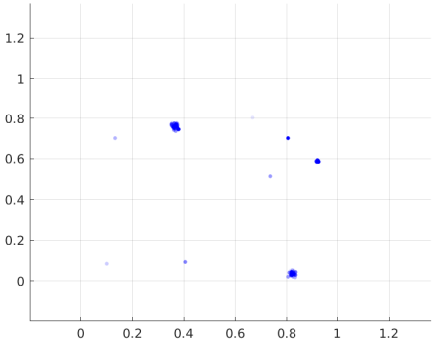

Figure 27: Nonstationary t=3102

Sequential

Sequential is a special subset of nonstationary cases. The overall image of the datasets changes and clusters appear and disappear sequentially. In Figures 28 to 31 a sequence of a sequential dataset is depicted in various moments of the simulated time. Figure 28 shows one cluster $C1(0.4, 0.2)$ and Figure 23 shows how $C1$ is already disappearing while $C2(0, 0.6)$ emerges. The disappearing of a cluster is implicitly shown by the fading blue points to visualize older data points. In Figure 30 cluster $C2$ is fading while cluster $C3(1, 0.9)$ emerges.

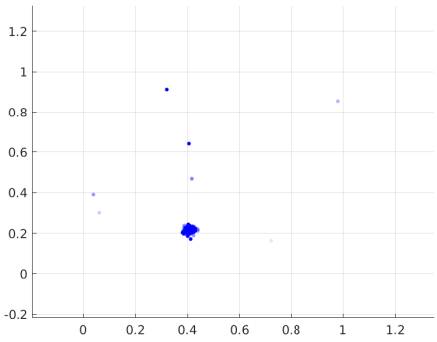

Figure 28: Sequential clusters t=867

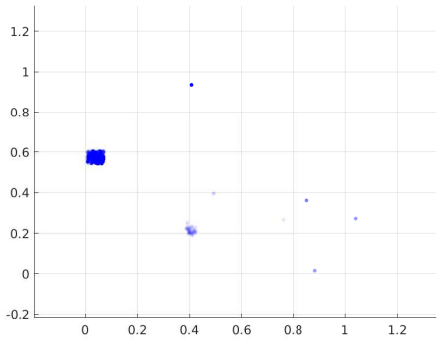

Figure 29: Sequential clusters t=1659

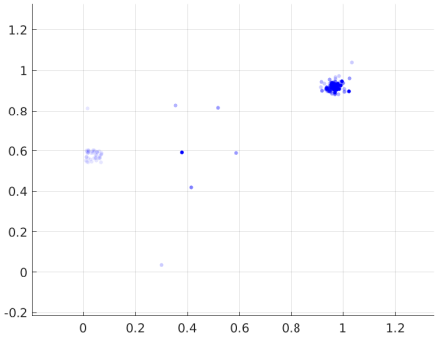

Figure 30: Sequential clusters t=2238

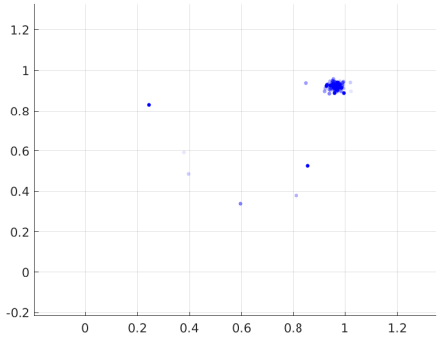

Figure 31: Sequential clusters t=3405

4.1.2 Input space size

Another configuration parameter around which we emphasize our experiments is the type of input space. Figures 32 and 33 show the two available types, tight and extensive. In a tight input space, clusters are much closer together and the ratio between the cluster diameter and the distance to a neighboring cluster is big. Visually space appears to be much more cluttered. On the other hand in the case of an extensive space, clusters appear to be far away from each other. In Figure 33 clusters are barely visible as they appear to be tiny since the ratio between cluster diameter and cluster distance is very small.

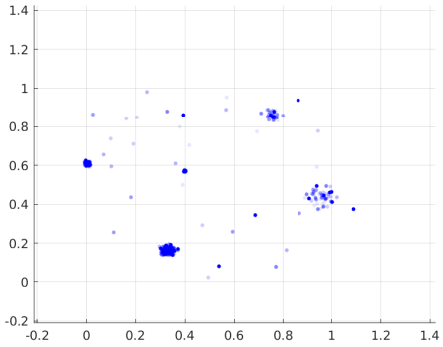

Figure 32: Tight input space

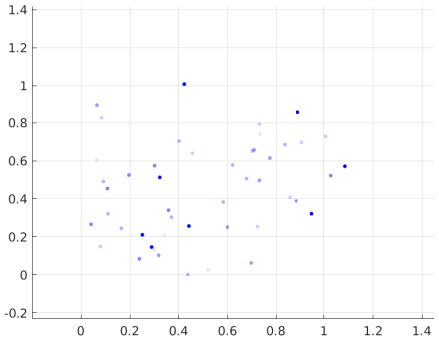

Figure 33: Extensive input space

4.1.3 Moving clusters

Lastly, we discuss the parameter to configure cluster movement. As stated before, nonstationarity and cluster movement generate different scenarios of concept drift.
An example for moving clusters is visible in the Figures 34 to 37. The cluster centroids change their geometric location as time passes and thus the surrounding distribution of cluster data points migrates with the centroid.

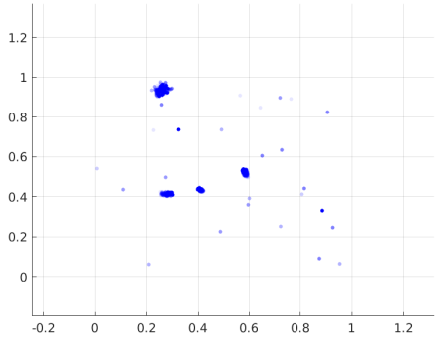

Figure 34: Moving clusters t=1000

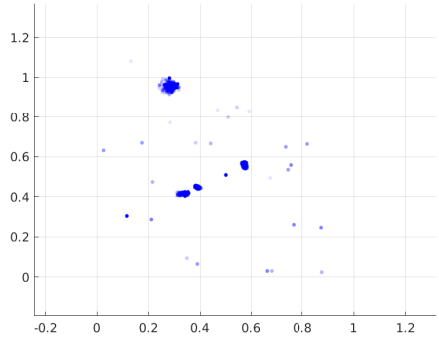

Figure 35: Moving clusters t=2000

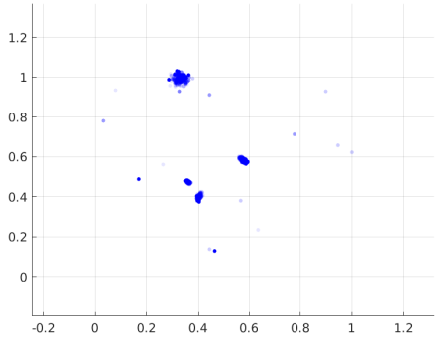

Figure 36: Moving clusters t=3000

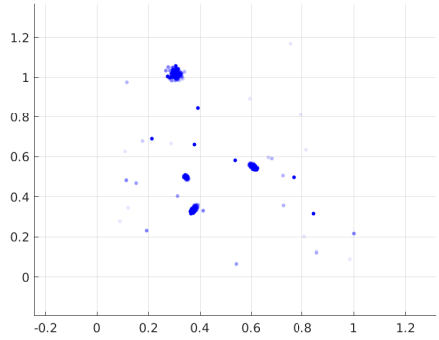

Figure 37: Moving clusters t=4000

4.2 Description of the datasets

Using the MDCStream wrapper as described in Section 3.3 we generated a set of datasets to test and evaluate classification algorithms. In Table 11 an overview of the configuration is listed for each scenario type. The three main parameters that are being varied are written in bold letters, namely the stationarity, space and moving clusters. The other configuration options remain consistent throughout the datasets. To summarize shortly the common configuration options, each scenario is created 20 times resulting in a total of 240 datasets. Each dataset consists of 10.000 instances whereas the first 1.000 are considered as training instances and are not part of the evaluation. Dimensions are randomly chosen between 2 and 100, outliers between 5% to 15% and clusters between 3 and 10. The cluster shapes are selected randomly between radial and multivariate

using MDCGens distribution functions. Additionally, the cluster densities vary and cluster overlap is avoided.

Dataset name	*sty-small-stc*	*seq-small-stc*	*nos-small-stc*	*sty-big-stc*	*seq-big-stc*	*nos-big-stc*	*sty-small-dyn*	*seq-small-dyn*	*nos-small-dyn*	*sty-big-dyn*	*seq-big-dyn*	*nos-big-dyn*
stationarity	stat.	seq.	nons.	stat.	seq.	nons.	stat.	seq.	nons.	stat.	seq.	nons.
space	tight	tight	tight	ext.	ext.	ext.	tight	tight	tight	ext.	ext.	ext.
movingClus	no	no	no	no	no	no	all	all	all	all	all	all
datasets	20	20	20	20	20	20	20	20	20	20	20	20
dimensions	many	many	many	many	many	many	many	many	many	many	many	many
outliers	med.	med.	med.	med.	med.	med.	med.	med.	med.	med.	med.	med.
clusters	few	few	few	few	few	few	few	few	few	few	few	few
densityDiff	many	many	many	many	many	many	many	many	many	many	many	many
space	tight	tight	tight	ext.	ext.	ext.	tight	tight	tight	ext.	ext.	ext.
overlap	no	no	no	no	no	no	no	no	no	no	no	no

Table 11: *MDCStream wrapper configuration for generating experimental datasets*

4.3 MOA algorithms configuration

The configuration used for the MOA algorithms can be seen in Table 12. The values were held constant for all experiments conducted.

	k	R	W	f	ρ
SimpleCOD	10	0.5	1000	—	—
MCOD	10	0.5	1000	—	—
ExactSTORM	10	0.5	1000	100	—
ApproxSTORM	10	0.5	1000	100	0.5
AbstractC	10	0.5	1000	—	—

Table 12: MOA Algorithm configuration used for the experiments.

4.4 Results and Discussion

The results obtained by the evaluation measures are depicted in the Figures 39 to 42 and a numeric overview may be seen in Table 13. The runtimes for each algorithm can be seen in Figure 38.
From the experimental results the following conclusions are drawn:

- **Classification performance**: The tested algorithms create similar classification results with MCOD slightly outperforming the others. In terms of

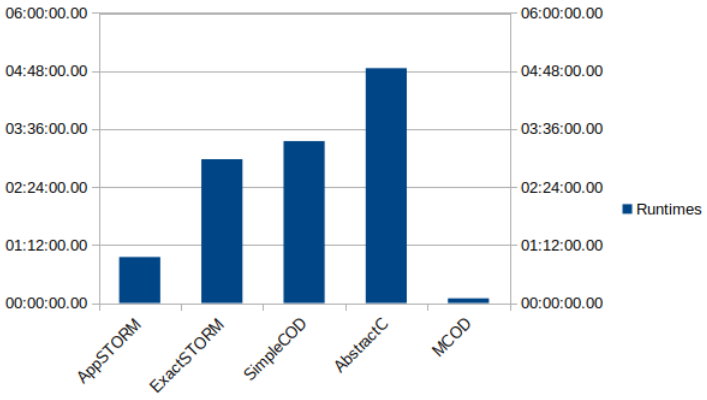

Figure 38: *Runtimes for each algorithm*

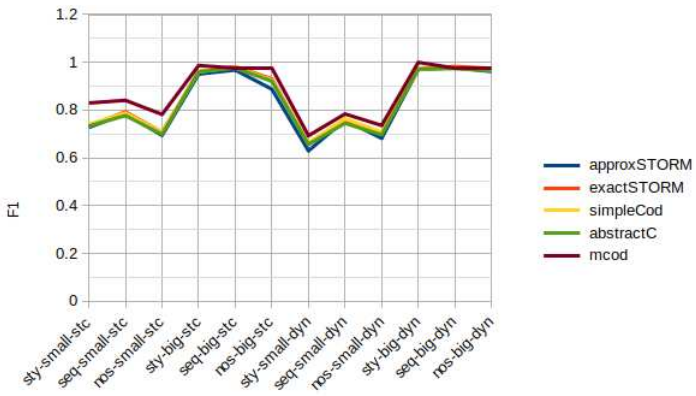

Figure 39: *F1 metric*

runtime performance, MCOD is much faster than the other algorithms as seen in Figure 38 which displays the runtimes for all 240 scenarios.
Similar classification performance results are expected since all the algorithms use the same underlying approach to extract outliers, namely the sliding window based approach using k-neighborhood. All algorithms have their optimization strategies for the problem, whereas the approach by MCOD stands out. MCOD groups points into micro clusters and reduces the number of distance calculations as thoroughly explained in Section 3.4. ApproxSTORM, the second-fastest algorithm, is about three times as fast as ExactSTORM. ApproxSTORM has an advantage over ExactSTORM by introducing two concepts to improve performance. $a)$ reduction of the

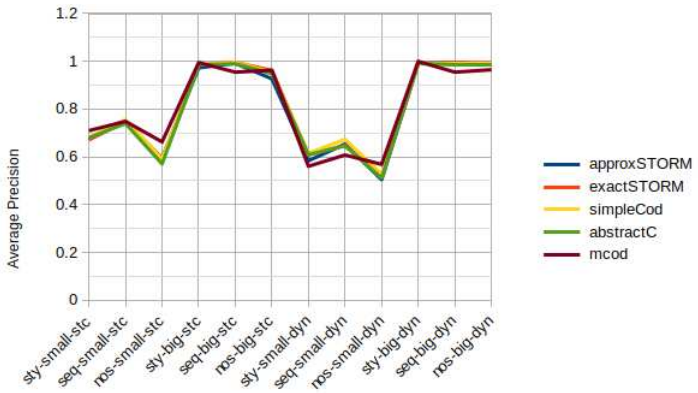

Figure 40: *Average precision metric*

ISB and b) reducing the node size as described in Section 3.4. Using these abstractions the amount of memory and range queries are reduced compared to ExactSTORM. Likewise, for the continuous outlier detectors COD, MCOD outperforms SimpleCOD by substantially reducing the number of range queries. The approach of AbstractC for k-nearest neighbor using the lifetime neighbor count results in the worst execution times from the examined algorithms.

- **Metrics**: The used metrics show complementary results when comparing the Figures 39 to 42. Metrics become redundant in this respect and therefore prove and solidify the results of our experiments.

- **Stationarity**: From the results it can be inferred that scenarios using various stationarities as described in Section 4.1.1 are no particular challenge for the algorithms under test. By taking a closer look the sequential scenario slightly outperforms the stationary and nonstationary case. All the metrics show slightly higher precision for the sequential case. This phenomenon occurs since there is only one cluster visible during every moment in time and therefore the input space appears less cluttered. Outliers are less likely to be situated close to a cluster and thus are easier to be classified as such.

- **Moving clusters**: The cluster movement from the test scenarios used did not have any impact on the classification results. However, this conclusion has to be drawn with care since it strongly depends on the configuration used. The clusters from our test scenarios were moving slowly relative to the used R value. Faster moving clusters would become a classification problem since not enough data points would accumulate within a specific radius to be classified as inliers.

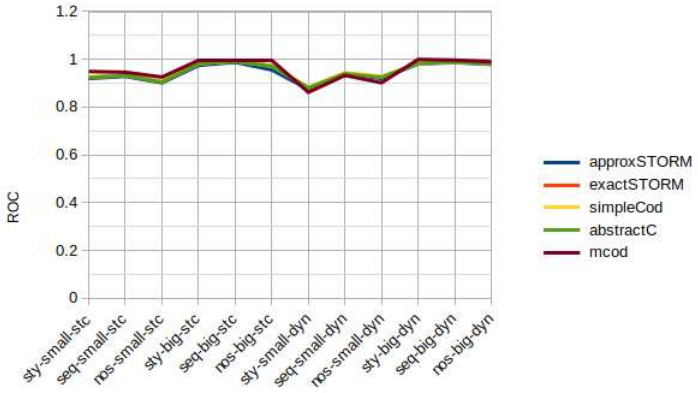

Figure 41: *ROC metric*

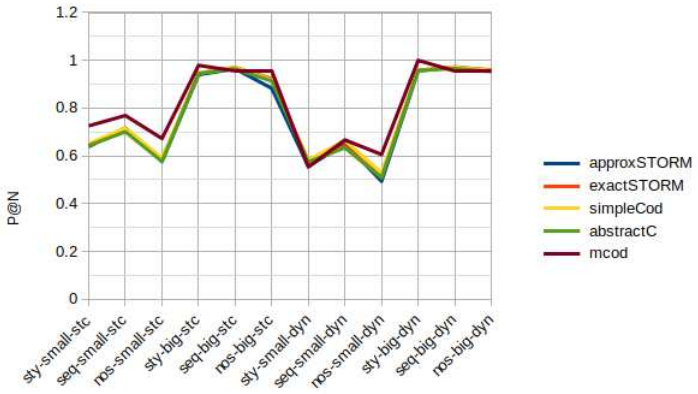

Figure 42: *Precision at N metric*

- **Space**: The most obvious factor influencing the classification results was the type input space. The tested outlier detection algorithms performed nearly perfect in extensive spaces while losing accuracy in tight spaces. Similar to the stationarity argument from above this behavior can be inferred due to the fact that a tight input space is more cluttered. Clusters are much closer together and the outliers in between are much closer to clusters. As consequence outliers are much more likely to be taken as inliers and vice versa. In extensive space clusters and outliers are far away from each other and within a specific radius R, algorithms are much more likely to find only outliers or clusters.

- **Algorithm configuration**: As stated in Section 3.4 there are three main

configuration parameters for the algorithms, window size W, k number of neighbors and the radius R within which data points are classified as neighbors. From the experimental evaluation and fine-tuning we inferred that increasing the value of W leads to better results because there are more data points in the currently observed window and thus more instances to be taken as a reference. However, increasing W drastically impacts the computational speed since more data points need to be kept in memory and more distance calculations need to be carried out. k parameter shows to be robust over a wider range though selecting too large values for k leads to worse precision results as clusters contain too little data points to be classified as such. The most important and noticeable changes during experiments were obtained by adapting the value of R. In tight input spaces larger R values produced better results while simultaneously reducing the quality of the results for extensive spaces. Generally speaking, the configuration is a weak spot for the algorithms to be used for real-life classification problems. To be able to obtain good classification results some a priory knowledge is needed to correctly configure the algorithms. Another problem may arise due to the use of the sliding window which permits the algorithms only a brief view of the data based on the total time of the dataset. Certain applications might need the algorithms to consider a global view on the data. Using the restricting window the cluster densities are artificially reduced by the algorithm which may lead to deceptive results.

Set	Algorithm	MaxF1	P@n	AP	ROC-AUC
sty-small-stc	ApproxSTORM	0.73 ± 0.24	0.64 ± 0.29	0.67 ± 0.28	0.92 ± 0.10
	ExactSTORM	0.73 ± 0.24	0.64 ± 0.28	0.67 ± 0.28	0.92 ± 0.09
	SimpleCOD	0.74 ± 0.23	0.65 ± 0.28	0.68 ± 0.28	0.93 ± 0.09
	AbstractC	0.73 ± 0.23	0.64 ± 0.28	0.68 ± 0.28	0.92 ± 0.09
	MCOD	0.83 ± 0.19	0.73 ± 0.33	0.71 ± 0.37	0.95 ± 0.10
seq-small-stc	ApproxSTORM	0.78 ± 0.23	0.72 ± 0.29	0.75 ± 0.30	0.93 ± 0.08
	ExactSTORM	0.79 ± 0.22	0.71 ± 0.30	0.75 ± 0.32	0.93 ± 0.08
	SimpleCOD	0.79 ± 0.22	0.72 ± 0.29	0.76 ± 0.31	0.93 ± 0.08
	AbstractC	0.78 ± 0.22	0.70 ± 0.30	0.74 ± 0.32	0.93 ± 0.08
	MCOD	0.84 ± 0.18	0.77 ± 0.26	0.75 ± 0.30	0.95 ± 0.09
nos-small-stc	ApproxSTORM	0.69 ± 0.25	0.58 ± 0.32	0.59 ± 0.33	0.90 ± 0.12
	ExactSTORM	0.71 ± 0.25	0.59 ± 0.33	0.59 ± 0.34	0.91 ± 0.11
	SimpleCOD	0.71 ± 0.25	0.59 ± 0.32	0.59 ± 0.34	0.91 ± 0.11
	AbstractC	0.70 ± 0.25	0.58 ± 0.32	0.57 ± 0.34	0.91 ± 0.11
	MCOD	0.78 ± 0.22	0.67 ± 0.33	0.66 ± 0.37	0.93 ± 0.13
sty-big-stc	ApproxSTORM	0.95 ± 0.14	0.94 ± 0.15	0.97 ± 0.10	0.98 ± 0.06
	ExactSTORM	0.96 ± 0.10	0.95 ± 0.13	0.99 ± 0.04	0.98 ± 0.04
	SimpleCOD	0.96 ± 0.10	0.95 ± 0.13	0.99 ± 0.04	0.98 ± 0.04
	AbstractC	0.96 ± 0.10	0.95 ± 0.13	0.99 ± 0.05	0.98 ± 0.04
	MCOD	0.99 ± 0.05	0.98 ± 0.09	1.00 ± 0.02	1.00 ± 0.02
seq-big-stc	ApproxSTORM	0.97 ± 0.08	0.96 ± 0.08	0.99 ± 0.02	0.99 ± 0.03
	ExactSTORM	0.98 ± 0.04	0.97 ± 0.07	1.00 ± 0.01	0.99 ± 0.02
	SimpleCOD	0.98 ± 0.05	0.97 ± 0.07	1.00 ± 0.01	0.99 ± 0.02
	AbstractC	0.98 ± 0.05	0.96 ± 0.07	0.99 ± 0.01	0.99 ± 0.02
	MCOD	0.98 ± 0.01	0.96 ± 0.03	0.95 ± 0.03	1.00 ± 0.00
nos-big-stc	ApproxSTORM	0.89 ± 0.21	0.88 ± 0.22	0.93 ± 0.15	0.96 ± 0.08
	ExactSTORM	0.93 ± 0.14	0.92 ± 0.15	0.96 ± 0.09	0.97 ± 0.05
	SimpleCOD	0.93 ± 0.14	0.92 ± 0.16	0.96 ± 0.10	0.97 ± 0.05
	AbstractC	0.92 ± 0.14	0.91 ± 0.16	0.95 ± 0.10	0.97 ± 0.05
	MCOD	0.97 ± 0.01	0.95 ± 0.02	0.96 ± 0.02	0.99 ± 0.01
sty-small-dyn	ApproxSTORM	0.63 ± 0.30	0.56 ± 0.34	0.58 ± 0.35	0.87 ± 0.13
	ExactSTORM	0.66 ± 0.29	0.57 ± 0.34	0.61 ± 0.35	0.88 ± 0.13
	SimpleCOD	0.66 ± 0.29	0.58 ± 0.33	0.61 ± 0.35	0.88 ± 0.13
	AbstractC	0.66 ± 0.29	0.57 ± 0.33	0.61 ± 0.35	0.88 ± 0.13
	MCOD	0.69 ± 0.29	0.55 ± 0.42	0.56 ± 0.46	0.86 ± 0.18
seq-small-dyn	ApproxSTORM	0.76 ± 0.22	0.65 ± 0.30	0.65 ± 0.32	0.94 ± 0.07
	ExactSTORM	0.75 ± 0.22	0.64 ± 0.30	0.65 ± 0.32	0.94 ± 0.06
	SimpleCOD	0.76 ± 0.22	0.66 ± 0.31	0.67 ± 0.33	0.94 ± 0.06
	AbstractC	0.74 ± 0.22	0.63 ± 0.30	0.65 ± 0.32	0.94 ± 0.06
	MCOD	0.78 ± 0.19	0.67 ± 0.30	0.61 ± 0.33	0.93 ± 0.09
nos-small-dyn	ApproxSTORM	0.68 ± 0.20	0.49 ± 0.29	0.51 ± 0.30	0.92 ± 0.07
	ExactSTORM	0.70 ± 0.19	0.51 ± 0.29	0.52 ± 0.31	0.93 ± 0.07
	SimpleCOD	0.71 ± 0.19	0.53 ± 0.29	0.53 ± 0.32	0.93 ± 0.07
	AbstractC	0.70 ± 0.18	0.51 ± 0.28	0.51 ± 0.30	0.93 ± 0.07
	MCOD	0.74 ± 0.23	0.61 ± 0.34	0.57 ± 0.38	0.90 ± 0.16
sty-big-dyn	ApproxSTORM	0.97 ± 0.08	0.96 ± 0.11	0.99 ± 0.03	0.98 ± 0.05
	ExactSTORM	0.97 ± 0.07	0.96 ± 0.11	0.99 ± 0.03	0.98 ± 0.04
	SimpleCOD	0.97 ± 0.08	0.96 ± 0.11	0.99 ± 0.03	0.98 ± 0.04
	AbstractC	0.97 ± 0.08	0.96 ± 0.11	0.99 ± 0.03	0.98 ± 0.04
	MCOD	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
seq-big-dyn	ApproxSTORM	0.98 ± 0.07	0.97 ± 0.08	0.99 ± 0.02	0.99 ± 0.04
	ExactSTORM	0.98 ± 0.05	0.97 ± 0.08	1.00 ± 0.02	0.99 ± 0.03
	SimpleCOD	0.98 ± 0.07	0.97 ± 0.08	0.99 ± 0.02	0.99 ± 0.03
	AbstractC	0.97 ± 0.07	0.97 ± 0.08	0.99 ± 0.02	0.99 ± 0.03
	MCOD	0.98 ± 0.01	0.95 ± 0.02	0.96 ± 0.02	1.00 ± 0.00
nos-big-dyn	ApproxSTORM	0.96 ± 0.09	0.96 ± 0.10	0.99 ± 0.04	0.98 ± 0.05
	ExactSTORM	0.97 ± 0.06	0.96 ± 0.09	0.99 ± 0.02	0.98 ± 0.04
	SimpleCOD	0.97 ± 0.06	0.96 ± 0.09	0.99 ± 0.02	0.98 ± 0.04
	AbstractC	0.97 ± 0.06	0.95 ± 0.09	0.99 ± 0.02	0.98 ± 0.04
	MCOD	0.98 ± 0.04	0.95 ± 0.06	0.96 ± 0.03	0.99 ± 0.04

Table 13: Experiment results.

5 Conclusion

This thesis aims to illustrate that a dataset generator able to produce dedicated test data scenarios is indispensable for the evaluation of stream classification algorithms. To do so we developed MDCStream, a highly configurable MATLAB tool for the generation of artificial multi-dimensional stream data with a focus on implementing different types of concept drift and nonstationarity. To show the potential of our dataset generator we evaluated five outlier detection algorithms contained in the popular open-source framework MOA. We tested the classification algorithms on 240 generated datasets which cover twelve different test scenarios focusing on *stationarity*, *cluster movement* and *input space size* with respect to intra-cluster distances. All datasets were configured with a variable number of dimensions and clusters, variable cluster shapes and densities and between 5% to 15% of outliers.
Referring back to the state of the art Section 2 we have summarized that it is very difficult to find a suitable and dedicated general-purpose dataset generator. At the same time artificially generated datasets are essential in the development and evaluation of classification algorithms to allow exhaustive testing covering even remote corner cases.
Using the generated datasets with MDCStream we did a comparative evaluation on MOA's outlier detection algorithms concluding that MCOD is the most suitable for applications in real live stream data fields. MCOD outperformed the other algorithms massively in terms of runtime performance and ever so slightly in the prediction accuracy. In general, the algorithms predictive performance shows not to be bothered by cluster movement given the limiting factor of a priory configuration. Most prominent deterioration proved to be witnessed during variations in the input space size. Tight and more cluttered input spaces were more challenging for the algorithms due to smaller density differences in the data points and the proximity of clusters and outliers. Scenarios using different stationarities validated the input space size argument. In the sequential case clusters appear one after the other and thus only one cluster is visible at a time. This arrangement produced slightly better classification performances.
Returning to MDCStream, it would have been very hard to do this comparative evaluation of MOA's algorithms, without the help of a highly configurable dataset generator. Using MDCStream it was simple to define test cases and to generate datasets for the analysis. The possibilities and advantages of having a good dataset generator, not only for the evaluation but also for the development of new classification algorithms, are immense. As the field of stream data analysis evolves there are lots of open issues to be addressed in the future and many possible features to be added to stream data generators like clusters with varying densities over time, datasets with varying dimensionalities and many more. Tools

like the MDCStream generator will become invaluable in the future of stream data analysis and evaluation of classification algorithms.

6 References

References

[1] Richard Johnson: Matlab Style Guidelines 2.0. March 2014

[2] IEEE Std 830-1993: IEEE Recomended Practice for Software Requirements Specifications, Dec 1993

[3] John Poland: "Three Different Algorithms for Generating Uniformly Distributed Random Points on the N-Sphere" Oct 24, 2000

[4] Fabrizio Angiulli, Fabio Fassetti: Detecting Distance-Based Outliers in Streams of Data

[5] Di Yang, Elke A. Rundensteiner, Matthew O. Ward: Neighbor-Based Pattern Detection for Windows Over Streaming Data

[6] Félix Iglesias, Tanja Zseby, Daniel Ferreira, Arthur Zimek: "MDCGen: Multidimensional Dataset Generator for Clustering", April 2019

[7] D. M. Hawkins: "Identification of outliers". Chapman and Hall London; New York, 1980

[8] Hodge, V. J.; Austin, J. "A Survey of Outlier Detection Methodologies", 2004

[9] Pooja Thakkar, Jay Vala, Vishal Prajapati: "Survey on Outlier Detection in Data Stream", February 2016

[10] Z. He, X. Xu, and S. Deng: “Discovering cluster-based local outliers”, Jun. 2003

[11] Fabrizio Angiulli, Fabio Fassetti: “Detecting Distance-Based Outliers in Streams of Data”, 2007

[12] Charu C. Aggarwal: “Data Mining: The Textbook”, Springer, 2015

[13] Félix Iglesias Vasquez, Tanja Zesby, Arthur Zimek: "Outlier Detection Based on Low Density Models", 2018

[14] Dragoljub Pokrajac, Aleksandar Lazarevic, Longin Jan Latecki: "Incremental Local Outlier Detection for Data Streams", April 2007

[15] Albert Bifet, Ricard Gavaldà, Geoff Holmes, Bernhard Pfahringer: "Data Stream Mining, A Practical Approach", May 2011

[16] Lab HNA. "streamDM." URL https://github.com/huawei-noah/streamDM., 2015

[17] Zhang, Haopeng, Yanlei Diao, and Neil Immerman. "Recognizing patterns in streams with imprecise timestamps." Proceedings of the VLDB Endowment 3.1-2 (2010): 244-255 URL http://avid.cs.umass.edu/sase/index.php?page=home, 2014

[18] P. Srivats. extbf Ostinato Packet Generator. URL https://ostinato.org, Aug 2019

[19] "KDG (Kinesis Data Generator)." URL https://awslabs.github.io/amazon-kinesis-data-generator/web/help.html, Jan 2019

[20] Squires M (2017). "fajer.js." URL https://github.com/marak/Faker.js/, Jun 2019

[21] Anand Narasimhamurthy, Ludmila I. Kuncheva: "A framework for generating data to simulate changing environments", 2007

[22] Patrick Lindstrom, Sarah Jane Delany, Brian Mac Namee: "Autopilot: Simulating Changing Concepts in Real Data"

[23] LMU Munich, On the Evaluation of Unsupervised Outlier Detection: http://www.dbs.ifi.lmu.de/research/outlier-evaluation/DAMI/

[24] E. M. Knorr and R. T. Ng, "Algorithms for mining distance-based outliers in large datasets," 1998, pp. 392– 403.

[25] J. Tang, Z. Chen, A. W.-c. Fu, and D. W. Cheung, "Enhancing effectiveness of outlier detections for low density patterns," 2002, pp. 535–548.

[26] S. Papadimitriou, H. Kitagawa, P. B. Gibbons, and C. Faloutsos, "LOCI: fast outlier detection using the local correlation integral," 2003, pp. 315–326.

[27] V. Barnett and T. Lewis: "Outliers in statistical data" 2nd ed. John Wiley & Sons Ltd, 1978

[28] G. Danuser and M. Stricker: "Parametric model fitting: from inlier characterization to outlier detection," vol. 20, no. 3, pp. 263–280, 1998.

[29] C. Leys, C. Ley, O. Klein, P. Bernard, and L. Licata: "Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median," Journal of Experimental Social Psychology, vol. 49, no. 4, pp. 764–766, Jul. 2013

[30] P. J. Rousseeuw and A. M. Leroy, Robust Regression and Outlier Detection. New York, NY, USA: John Wiley & Sons, Inc., 1987.

[31] M. Goldstein and A. Dengel, "Histogram-based outlier score (HBOS): A fast unsupervised anomaly detection algorithm," in KI-2012: Poster and Demo Track, 2012, pp. 59–63.

[32] Amardeep Kaur, Dr. M.P.S. Bhatia, Dr. S.M. Bhaskar: "State of the art of outlier detection in streaming data", 2007

[33] Ji Zhang: "Advancements of Outlier Detection: A Survey", 2013

[34] Fabrizio Angiulli, Fabio Fassetti: "Distance-based outlier queries in data streams:the novel task and algorithms", 2010

[35] Leigh Metcalf, William Casey: "Cybersecurity and Applied Mathematics" cap 2.10.1, 2016

[36] Image based on: Indre Zliobaite: "Learning under Concept Drift: an Overview". Section 2.2 page 7. Computing Research Repository (arXiv: CoRR) abs/1010.4784. arXiv:1010.4784 URL http://arxiv.org/abs/1010.4784, 2010

[37] Gerhard Widmer, Miroslav Kubat: "Learning in the Presence of Concept Drift and Hidden Contexts" 1996

[38] David M W Powers: "Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness and Correlation", 2010

[39] Guilherme O. Campos, Arthur Zimek, Jörg Sander, Ricardo J. G. B. Campello, Barbora Micenková, Erich Schubert, Ira Assent 4 Michael E. Houle: "On the evaluation of unsupervised outlier detection: measures, datasets, and an empirical study", 2016

[40] Maria Kontaki, Anastasios Gounaris, Apostolos N. Papadopoulos, Kostas Tsichlas, Yannis Manolopoulos: "Continuous Monitoring of Distance-Based Outliers over Data Streams", 2011

[41] Alexey Tsymbal: "The problem of concept drift: definitions and related work" Department of Computer Science Trinity College Dublin, Ireland, 2004

[42] Indrė Žliobaitė, Mykola Pechenizkiy, João Gama: "An overview of concept drift applications"

[43] Bifet, A., Holmes, G., Kirkby, R., and Pfahringer: B. "Moa: Massive online analysis. Journal of Machine Learning Research" 11:1601–1604, 2010

[44] Image based on: Stefanowski J., Brzezinski D. "Stream Classification" page 3 2017 In: Sammut C., Webb G.I. (eds) Encyclopedia of Machine Learning and Data Mining. Springer, Boston, MA

[45] Krempl, G., Žliobaitė, I., Brzezinski, D., Hüllermeier, E., Last, M., Lemaire, V., Noack, T., Shaker, A., Sievi, S., Spiliopoulou, M., and Stefanowski: "Open challenges for data stream mining research" Explorations, 16(1):1–10, 2014

[46] Image based on Fig.9 "ROC curve averaging" from Tom Fawcett: "An introduction to ROC analysis" 2005

A Appendix

A.1 MDCGen example

In this subsection an example of a MDCGen Matlab script is shown in Listing 5 and the respective result is depicted in Figure 43.

```
warning on
warning('backtrace', 'off');
addpath(genpath('config_build/src/'));
addpath(genpath('mdcgen/src'));

config.seed = 3;
config.nDatapoints = 2000;
config.nDimensions = 3;
config.nClusters = 6;
config.nOutliers = 100;
config.compactness = 0.3;
config.rotation = 1;
config.distribution = [6 1 2 4 3 5];

[ result ] = mdcgen( config );
scatter3(result.dataPoints(:,1), ...
         result.dataPoints(:,2), ...
         result.dataPoints(:,3), ...
         5, result.label,'fill');
grid on
axis([0 1 0 1 0 1])
```

Listing 5: MDCGen example Matlab script

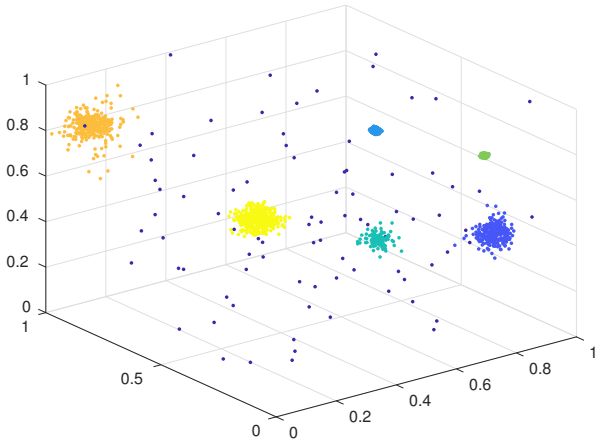

Figure 43: *MDCGen example*

A.2 MDCGen and MDCStream example

In this subsection an example of a MDCGen and MDCStream Matlab script is shown in Listing 6 and the respective result is depicted in the Figures 44 to 49. The fading blue points illustrate older data points.

```
warning on
warning('backtrace', 'off');

addpath(genpath('config_build/src'));
addpath(genpath('mdcstream/src'));
addpath(genpath('display'));
addpath(genpath('data_provider'));

fprintf('Creating MDCGen dataset ... \n');
p.seed = 1;
config.seed = 15;
config.nDatapoints = 2000;
config.nDimensions = 2;
config.nClusters = 5;
config.nOutliers = 100;
config.distribution = [1 6 2 6 1];
config.compactness = [0.7 0.3 0.5 0.5 0.6];
data = mdcgen( config );

fprintf('Creating MDCStream dataset ... \n');
config.stationary = 0;
config.mu = [20 2 1 3 2 3];
config.startTime = [0 300 0 500 100 0];

data = mdcstream(data, config);
displayFade(data, 100);
```

Listing 6: MDCGen and MDCStream example Matlab script

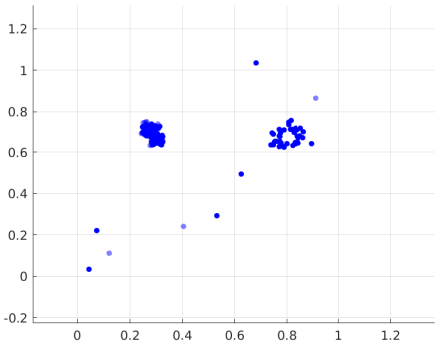

Figure 44: MDCStream t=552

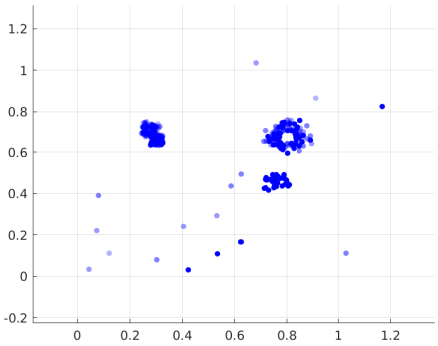

Figure 45: MDCStream t=1084

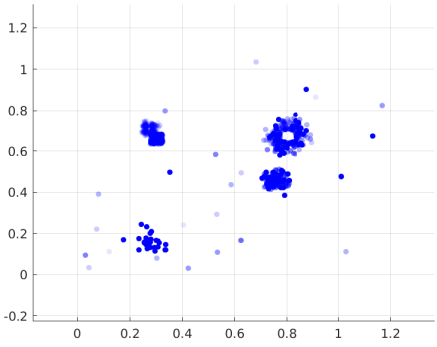

Figure 46: MDCStream t=1232

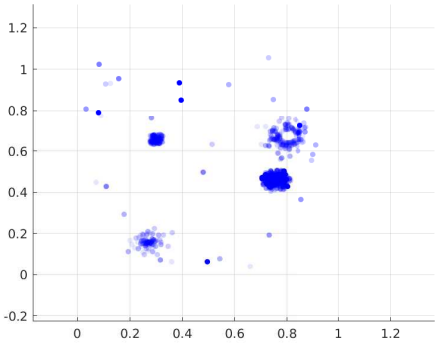

Figure 47: MDCStream t=1768

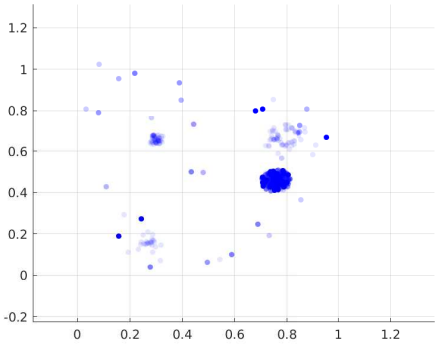

Figure 48: MDCStream t=2009

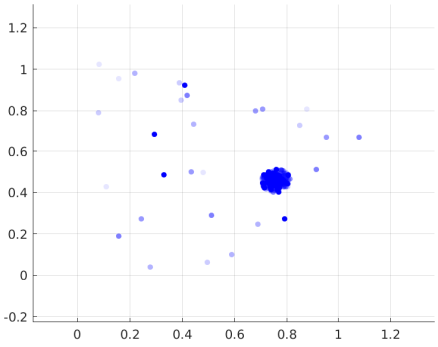

Figure 49: MDCStream t=2887

A.3 MDCStream wrapper configuration

In this subsection an example of a Matlab script for the MDCStream wrapper is shown in Listing 7.

```
1
2  clear
3  addpath(genpath('src'));
4
5  p.seed            = 15;
6  p.nDatapoints     = 10000;
7  p.scenarioName    = {'scenarioA' , 'scenarioB'  };
8  p.nOfDataSets     = {20          , 20           };
9  p.stationary      = {'stationary', 'sequential' };
10 p.space           = {'tight'     , 'extensive'  };
11 p.movingClusters  = {'all'       , 'no'         };
12 p.dimensions      = {'two'       , 'many'       };
13 p.densityDiff     = {'many'      , 'many'       };
14 p.outliers        = {'few'       , 'many'       };
15 p.clusters        = {'few'       , 'many'       };
16 p.overlap         = {'no'        , 'yes'        };
17
18 generateDataSets(p, 'dataRoot');
```

Listing 7: MDCStream-wrapper Matlab script

List of Figures

List of Tables