ERGEBNISSE AUS DER INFORMATIK

Recent advances in data collecting devices and data storage systems are continuously offering cheaper possibilities for gathering and storing increasingly bigger volumes of data. Similar improvements in the processing power and data bases enabled the accessibility to a large variety of complex data. Data mining is the task of extracting useful patterns and previously unknown knowledge out of this voluminous, various data. This thesis focuses on the data mining task of clustering, i.e. grouping objects into clusters such that similar objects are assigned to the same cluster while dissimilar ones are assigned to different clusters. While traditional clustering algorithms merely considered static data, today's applications and research issues in data mining have to deal with continuous, possibly infinite streams of data, arriving at high velocity. Web traffic data, click streams, surveillance data, sensor measurements, customer profile data and stock trading are only some examples of these daily-increasing applications.

Since the growth of data sizes is accompanied by a similar raise in their dimensionalities, clusters cannot be expected to completely appear when considering all attributes together. Subspace clustering is a general approach that solved that issue by automatically finding the hidden clusters within different subsets of the attributes rather than considering all attributes together.

In this thesis, novel methods for an efficient subspace clustering of high-dimensional data streams are presented and deeply evaluated. Approaches that efficiently combine the anytime clustering concept with the stream subspace clustering paradigm are intensively studied. Additionally, efficient and adaptive density-based clustering algorithms are presented for high-dimensional data streams. New algorithmic solutions for an energy-efficient in-sensor-network aggregation and a light-weighted clustering are presented for sensor streaming data. Novel open-source assessment framework and evaluation measures are presented for subspace stream clustering.
Primarily, efficient models of advanced and complex clustering tasks are for the first time contributed for data streams.

ISBN 978-3-86359-318-6

Marwan Hassani Efficient Clustering of Big Data Streams

Marwan Hassani

Efficient Clustering of Big Data Streams

Edition Wissenschaft Apprimus

Efficient Clustering of Big Data Streams

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Ingenieurwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Ingenieur
Marwan Hassani
aus Aleppo, Syrien

Berichter: Univ.-Prof. Dr. rer. nat. Thomas Seidl
Assoc. Prof. Dr. Mohamed M. Gaber
Prof. Dr.-Ing. Stefan Kowalewski

Tag der mündlichen Prüfung: 26.01.2015

To my family who supported me with patience and love. To my beautiful daughters Yumna and Amena, without whom this thesis would have been completed one year earlier.

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

Marwan Hassani:

Efficient Clustering of Big Data Streams

1. Auflage, 2015

Ergebnisse aus der Informatik
Band 4

Gedruckt auf holz- und säurefreiem Papier, 100% chlorfrei gebleicht.

Apprimus Verlag, Aachen, 2015
Wissenschaftsverlag des Instituts für Industriekommunikation und Fachmedien an der RWTH Aachen
Steinbachstr. 25, 52074 Aachen
Internet: www.apprimus-verlag.de, E-Mail: info@apprimus-verlag.de

Printed in Germany

ISBN 978-3-86359-318-6

D 82 (Diss. RWTH Aachen University, 2015)

Contents

Abstract

Recent advances in data collecting devices and data storage systems are continuously offering cheaper possibilities for gathering and storing increasingly bigger volumes of data. Similar improvements in the processing power and data bases enabled the accessibility to a large variety of complex data. Data mining is the task of extracting useful patterns and previously unknown knowledge out of this voluminous, various data. This thesis focuses on the data mining task of clustering, i.e. grouping objects into clusters such that similar objects are assigned to the same cluster while dissimilar ones are assigned to different clusters. While traditional clustering algorithms merely considered static data, today's applications and research issues in data mining have to deal with continuous, possibly infinite streams of data, arriving at high velocity. Web traffic data, click streams, surveillance data, sensor measurements, customer profile data and stock trading are only some examples of these daily-increasing applications.

Since the growth of data sizes is accompanied by a similar raise in their dimensionalities, clusters cannot be expected to completely appear when considering all attributes together. Subspace clustering is a general approach that solved that issue by automatically finding the hidden clusters within different subsets of the attributes rather than considering all attributes together.

In this thesis, novel methods for an efficient subspace clustering of high-dimensional data streams are presented and deeply evaluated. Approaches that efficiently combine the anytime clustering concept with the stream subspace clustering paradigm are intensively studied. Additionally, efficient and adaptive density-based clustering algorithms are presented for high-dimensional data streams. New algorithmic solutions for an energy-efficient in-sensor-network aggregation and a light-weighted clustering are presented for sensor streaming data. Novel open-source assessment framework and evaluation measures are presented for subspace stream clustering. Primarily, efficient models of advanced and complex clustering tasks are for the first time contributed for data streams.

Zusammenfassung

Aktuelle Entwicklungen in den Datenerfassungsgeräten und Datenspeichersystemen bieten ständig günstigere Möglichkeiten zur Sammlung und Speicherung von großen Datenmengen. Mit steigender Rechenleistung und effizienteren Datenbanken wird der Zugang zu einer Vielzahl komplexer Daten ermöglicht. Die Aufgabe des Data Mining ist das Extrahieren von nützlichen Mustern in diesen umfangreichen und unterschiedlichen Daten, um schließlich neue Erkenntnisse zu gewinnen. Diese Dissertation konzentriert sich auf die Clustering-Analyse, deren Ziel darin besteht, ähnliche Objekte in dieselben Cluster und unähnliche Objekte in verschiedene Cluster zu gruppieren. Während traditionelle Clustering-Algorithmen lediglich statische Daten betrachten, müssen heutige Algorithmen mit vielen, kontinuierlichen, möglicherweise unendlichen Datenströmen, die mit hoher Geschwindigkeit ankommen, umgehen.

Aufgrund der immer höheren Dimensionalität in aktuellen Anwendungen, liefern traditionelle Clustering-Algorithmen, unter Berücksichtigung aller Dimensionen, nur selten aussagekräftige Cluster. Ein allgemeiner Ansatz zur Lösung dieses Problems ist die Subspace-Clustering-Analyse. Anstatt alle Dimensionen gemeinsam zu berücksichtigen, werden Cluster automatisch in verschiedenen Teilräumen unterschiedlicher Dimensionalität gesucht.

In dieser Dissertation werden neue Methoden für die effiziente Subspace-Clustering-Analyse von hochdimensionalen Datenströme vorgestellt und mit dem Anytime-Paradigma kombiniert. Darüber hinaus werden effiziente und adaptive dichtebasierte Clustering-Algorithmen für hochdimensionale Datenströme entwickelt. Speziell für Sensordatenströme, werden neue algorithmische Lösungen für eine energieeffiziente netzwerkinterne Aggregation untersucht. Die in dieser Dissertation entwickelten Ansätze tragen maßgeblich zum aktuellen Forschungsstand im Bereich der effizienten Analyse von Datenströmen bei.

Chapter 1

Overview

1.1 Introduction

The huge recent advances in the data generation and storage systems resulted in cheap means for satisfying the eternal human eagerness to increasingly collect more data. As a natural consequence to the existence of these big amounts of data, an exponential growth of the data processing power, and huge advances of the data base management systems, took place recently. In spite of these advances, the big size and the high complexity of recent data exceeded the human ability of exploration and interpretation. This motivated the need for a collection of compression, selection, categorization and visualization tools that extract some knowledge from the data bases. These tools were organized into a process called KDD (Knowledge Discovery from Data bases) in [HKP06]. It consists of multiple steps as it is shown in Figure 1.1. The first step is the data cleaning and integration which basically removes noise and inconsistency from the data and combines multiple data bases into a data warehouse. As the integrated data is usually too general, the next step is to select the data that is relevant to the task at hand, and to perform some transformation by summarizing or aggregating it for instance. This preprocessed data is now ready for the next step, the *Data Mining* step, where intelligent methods are applied to extract some patterns. The last step evaluates these patterns and presents a comprehensible visualization of the mined data to the user to collect the aimed knowledge. While some of the techniques presented within this thesis are categorized under the *Evaluation and Visualization* step, the main scope of this thesis is the *Data Mining* step.

More precisely, we focus in this thesis on the *clustering* task of data mining. Clustering is a well-established data mining concept that aims at automatically

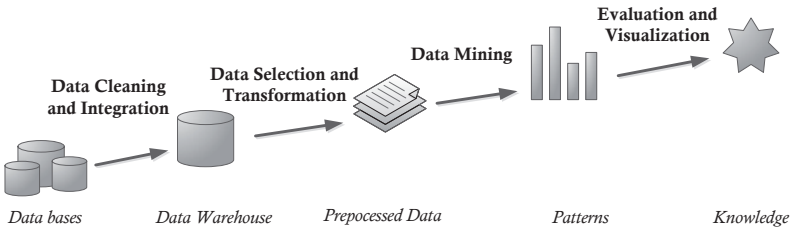


Figure 1.1: The process of Knowledge Discovery from Databases (KDD) as described in [HKP06].

grouping *similar* data objects while separating *dissimilar* ones. This process is strongly depending on the notion of *similarity*, which is often based on some distance measure. Thus, similar objects are usually close to each other while dissimilar ones are far from each other. The clustering task is performed without a previous knowledge of the data, or in an *unsupervised* manner. During the early stages of data mining research, the whole data objects were considered to be statically and permanently stored in the memory. This allowed the designed data mining technique to perform as much passages over the objects as needed to deliver the desired patterns. With the recent growth of the data size and the easiness of collecting data, these settings are not any more convenient. The size of the continuously generated data and the limited storage capacity allow in many scenarios for a single passage over the data, and users are interested in gaining a real time knowledge about the data as they are produced.

A data stream is an ordered sequence of objects that can be read once or very small number of times using limited processing and computing storage possibilities. This sequence of objects can be endless and flows usually at high speeds with a varying underlying distribution of the data. This fast and infinite flow of data objects does not allow the traditional permanent storage of the data and thus multiple passages are not any more possible. Many domains are dealing essentially with data streams. The most prominent examples include network traffic data, telecommunication records, click streams, weather monitoring, stock trading, surveillance data, health data, customer profile data, and sensor data. There is many more to come. A very wide spectrum of real-world streaming applications is expanding. Particularly in sensor data, such applications spread from home scenarios like the smart homes to environmental applications and monitoring tasks in the health sector [HS11, QBG+13], but do not end with military

applications. Actually, any source of information can easily be elaborated to produce a continuous flow of the data. Another emerging streaming data sources are social data. In a single minute, $546,000$ tweets are happening, $2,460,000$ pieces of content are shared on Facebook and 72 hours of new videos are uploaded to YouTube*. Users are interested of gaining the knowledge out of these information during their same minute of generation. A delay, say till the next minute, might result in an outdated knowledge.

Figure 1.2 gives some examples about real world application that produce data streams. Most of these scenarios are covered within the scope of this thesis. Figure 1.2(a) shows an example about wired streaming data that monitor some flowing phenomenon like network traffic data, click streams or airport camera monitoring. Figure 1.2(b) presents a visualization of streaming tweets with a certain tag and within a certain time using the *Streamgraph* framework [BW08]. Figure 1.2(c) depicts an application of a wireless sensor network deployment in a bridge for surveillance or load observation. Sensors are producing continuous streams of readings, and experts need to collect a real time knowledge about the stability of the bridge in the case of emergency, or gather regular reports in the normal case. Similarly, Figure 1.2(d) shows an example of sensors collecting temperature, humidity and light information from multiple offices in Intel Berkeley Research Lab [Dat04]. Such sensors are usually of limited storage, processing power and battery life. Figure 1.2(e) presents another type of sensor streaming data where an eye-tracking system is used to record the duration and the position of each eye fixation over the monitor during a human reading or writing process [VNT+14]. In Figure 1.2(f), a body sensor network is producing multiple streams about the health status of the runner. Other sensors are collecting streams of other contextual information like the weather and location information. These can be processed on a local mobile device or a remote server to gain, for instance, some knowledge about the near-future status [QBG+13].

Stream clustering aims at detecting clusters that are formed out of the evolving streaming objects. These clusters must be *continuously* updated as the stream emerges to follow the current distribution of the data. These clusters represent mainly the gained knowledge out of the clustering task. In this thesis, advanced stream clustering models are introduced. These models are mainly motivated by the basic challenges that we have observed for clustering of streaming data in real world scenarios, particularly sensor streaming data (cf. Figure 1.2). In the

*As on July 2014, Sources: domo.com and statisticbrain.com

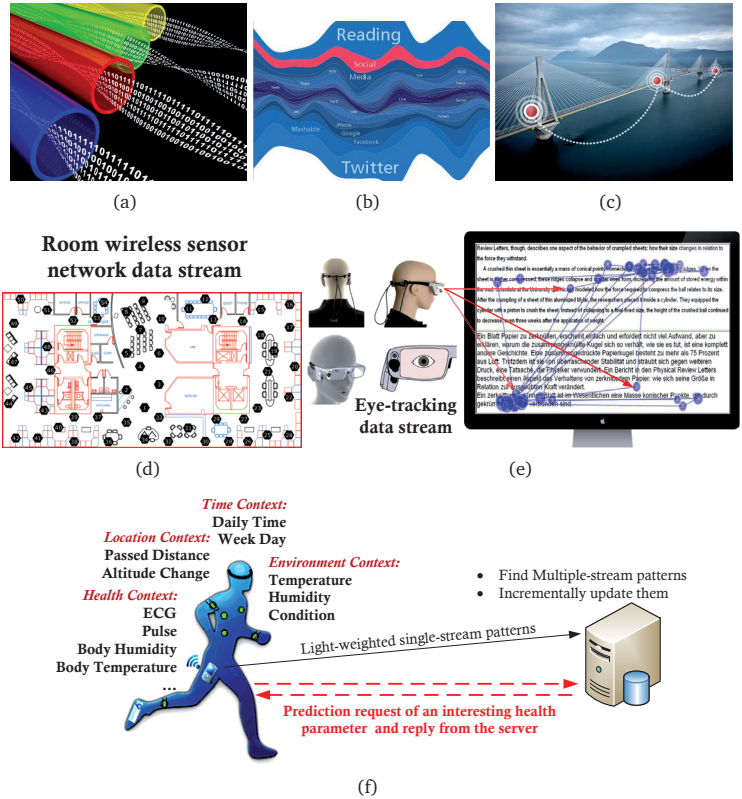


Figure 1.2: Examples about the data streams handled in this thesis.

following, we will list these challenges in Section 1.2. The contribution of the different approaches as well as the structure of this thesis are then presented in Section 1.3.

1.2 Challenges for Stream Clustering

Designing stream clustering approaches has some unique special challenges. We list in the following the different paradigms that make it challenging to design a stream clustering approach.

Adaptation to the Stream Changes, and Outlier Awareness:

The algorithm must incrementally cluster the stream data points to detect evolving clusters over the time, while forgetting outdated data. New trends of the data must be detected at the same time of their appearance. Nevertheless, the algorithm must be able to distinguish new trends of the stream from outliers. Fulfilling the up-to-date requirement contradicts the outlier awareness one. Thus, meeting this tradeoff is one of the basic challenges of any stream clustering algorithm.

Storage Awareness, and High Clustering Quality:

Due to the huge sizes and high speeds of streaming data, any clustering algorithm must perform as few passages over the objects as possible. In most cases, the application and the storage limitations allow only for a single passage. However, high quality clustering results are requested to make the desired knowledge out of the data stream. Most static clustering models tend to deliver an initial, sometimes random, clustering solution and then optimize it by revisiting the objects to maximize some similarity function. Although such multiple-passages possibility does not exist for streaming algorithms, the requirement of an optimized, high quality clustering does still exist.

Efficient Handling of High-Dimensional, Different-Density Streaming Objects:

The current huge increase of the sizes of data was accompanied with a similar boost in their number of dimensions. This applies of course to streaming data too. For such kinds of data with higher dimensions, distances between the objects grow more and more alike due to an effect termed *curse of dimensionality* [BGRS99]. According to this effect, applying traditional clustering algorithms in the full-space merely will result in considering almost all objects as outliers, as the distances between them grow exponentially with their dimensionality d. The latter fact motivated the research in the area of *subspace clustering* over static data in the last decade, which searches for clusters in all of the $2^d - 1$ subspaces of the data by excluding a subgroup of the dimensions at each step. Apparently this implies higher complexity of the algorithm even for static data, which makes it even more challenging when considering streaming data.

Additionally, as the stream evolves, the number, the density and the shapes of clusters may dramatically change. Thus, assuming a certain number of clusters like in k-means-based clustering models or setting a static density threshold as in the DBSCAN-based clustering models is not convenient for a stream clustering approach. A self-adjustment to the different densities of the data is strongly needed while designing a stream clustering algorithm. Again, this requirement is in conflict with the storage awareness necessity.

Flexibility to Varying Time Allowances Between Streaming Objects:

An additional, natural characteristic of data streams (e.g. sensor data) is the fluctuating speed rate. Streaming data objects arrive usually with different time allowances between them, although the application settings would assume a constant stream speed. Available stream clustering approaches, called budget algorithms in this context, strongly restrict their model size to handle minimal time allowance to be on the safe side. In the case of reduced stream speed, the algorithm remains idle during the rest of the time, till the next streaming object arrives. *Anytime mining algorithms*, designed recently for static data, try to make use of any given amount time to deliver some result. Longer given times, imply higher clustering quality. This idea was adopted for clustering streaming data. Although this setting can be seen as an opportunity for improving the clustering quality rather than a challenge, it is not trivial to have a flexible algorithmic model that is able to deliver some result even with very fast streams.

Energy-Awareness, and Lightweight Clustering of Sensor Data Streams:

Wireless sensor nodes are equipped with a small processing device, a tiny memory, a small battery in addition to the sensing unit [PSC05]. This encouraged the research in the area of in-sensor-network mining, where the nodes do some preprocessing of the sensed data instead of simply forwarding it. In many of these applications, sensor nodes are distributed in unreachable areas without a cheap possibility of changing the battery. Thus, the usability time of the node is bounded by the battery lifetime. In this manner, besides the previously mentioned challenges, clustering sensor streaming data has to carefully consume the processing and energy resources. In fact, clustering and aggregation approaches are used within wireless sensor networks to save energy by preprocessing the data in the node, and forwarding the relevant ones merely.

1.3 Contribution and Thesis Structure

In this thesis, we introduce novel, efficient stream clustering algorithms that consider all of the previous challenges. The thesis is structured in four main parts and a summary. In Part I, we present efficient methods for clustering sensor data and aggregating sensor nodes. In Part II, we present our novel high-dimensional density-based stream clustering techniques and in Part III, we introduce advanced anytime stream clustering approaches. In Part IV, we contribute our unique subspace stream clustering framework as well as the subspace cluster mapping evaluation measure. In all of the parts, the first and the second challenges mentioned in Section 1.2 were carefully considered. Each of the rest of the challenges was the main focus of the other parts, as we will explain in the corresponding part. In the following, we take a quick glance at the contents and the contribution of these chapters.

Part I

In the first chapter, we develop three novel methods for an energy-efficient in-sensor network aggregation of data. In the algorithms presented in this part, the main focus is the fifth challenge mentioned in Section 1.2.

In Chapter 2, we present the *EDISKCO* algorithm, an Energy-Efficient Distributed In-Sensor-Network K-Center Clustering algorithm with Outliers. Sensor networks have limited resources in terms of available memory and residual energy. As a dominating energy consuming task, the communication between the node and the sink has to be reduced for a better energy efficiency. Considering memory, one has to reduce the amount of stored information on each sensor node. The EDISKCO algorithm perform an outlier-aware k-center clustering over the sensed streaming data in each node and forwards the clustering result to the neighboring coordinator node. For that, the nodes consumes considerably less energy than the cost of forwarding all of its readings to the sink. The update from the node to the coordinator happens only upon a certain deviation of the cluster radii, controlled by the ϵ threshold, or upon the birth or the deletion of a cluster. One node is selected as a coordinator from each spatially correlated subgroup of the nodes, depending on the amount of the residual energy. The coordinator performs another k-center clustering over the multiple clustering solutions arriving from its neighboring nodes and forwards the solution to the far sink. These are major contributions as we perform a single passage over the data

by using a $O(k)$ storage over the nodes and getting finally a high clustering quality of a $(4+\epsilon)$-approximation to the optimal clustering. But the main contribution is performing up to 95% less communication tasks on the nodes compared to a state-of-the-art technique. Thus huge savings of energy are achieved.

In Chapter 3, a weighted version of EDISKCO, the *SenClu* algorithm, is presented. It guarantees a faster adaptation to the new trends of the drifting data streams. The technique gives more importance to new data points, while slowly forgetting older ones by giving them less weight. To achieve this, we contribute a novel, light-weighted decaying function that can be implemented on the tiny processing unit and works on the limited storage capacity of sensor nodes. SenClu achieves even better clustering quality than EDISKCO, while draining almost the same amount of energy. In Chapter 4, a further challenge for aggregating streaming data within the sensor network, is tackled. The physical clustering of sensor nodes depending on their similarity is considered in the presented *ECLUN* algorithm. The readings of a carefully selected representative node are used to simulate the measurements of similar nodes. While the recent approaches concentrated on the full-dimensionality correlation between the readings, ECLUN selects the representatives depending on the subspace correlation between some attributes of the measurements, as well as the spatial similarity. Additionally, we uniformly distribute the usage of energy between the nodes and cope with the cases of single-node clusters by changing representatives according to the residual energy. This results in a longer lifetime of the whole sensor network as nodes die close to each other.

Part II

In the second part, we develop two density-based stream clustering algorithms. Here, the third challenge mentioned in Section 1.2 is mainly considered.

In Chapter 5, a unique, efficient projected stream clustering algorithm is introduced for handling high-dimensional, noisy, evolving data streams. The chapter starts by explaining the subspace clustering concept used for high-dimensional data and then differentiating it from the projected clustering. The technique, *PreDeConStream*, is based on a two-phase model. The first phase represents the process of the online maintenance of data summaries, called microclusters, that is then passed to an offline phase for generating the final clustering. The technique works on incrementally updating the output of the online phase stored in a

microcluster structure. Taking those microclusters that are fading out over time into consideration speeds up the process of assigning new data points to existing clusters. The algorithm localizes the change to the previous clustering result, and smartly uses a clustering validity interval to make an efficient offline phase.

In Chapter 6, we contribute a hierarchical, self-adaptive, density-based stream clustering model. The *HASTREAM* algorithm presented in this chapter, focuses on smoothly detecting the varying number, densities and shapes of the streaming clusters. A cluster stability measure is applied over the summaries of the streaming data, to extract the most stable offline clustering. To improve the efficiency of the suggested model, some methods from the graph theory are adopted and others were contributed, to incrementally update a minimal spanning tree of microclusters. This tree is used to continuously extract the final clustering, by localizing the changes that appeared in the stream, and maintaining the affected parts merely.

Part III

Advanced anytime stream clustering approaches are contributed in the third part. By considering all other challenges, the main focuses of this part are the third and the fourth challenges mentioned in Section 1.2.

In Chapter 7, we present the *LiarTree* algorithm that provides precise stream summaries and effectively handles noise, drift and novelty at any given time. We prove that the runtime of the our anytime algorithm is logarithmic in the size of the maintained model opposed to a linear time complexity often observed in previous approaches. The main contributions of our technique are enabling the anytime concept to fast adapt to the new trends of the data, filtering noise and keeping a logarithmic complexity.

In Chapter 8, we add even another complexity dimension to the problem discussed in Chapter 7. The high-dimensionality paradigm is considered together with the varying arrival times and the streaming aspects of the data. A subspace anytime stream clustering algorithm, called *SubClusTree*, is presented in this chapter, that can flexibly adapt to the different stream speeds and makes the best use of available time to provide a high quality subspace clustering. It uses a compact index structures to maintain stream summaries in the subspaces in an online fashion. It uses flexible grids to efficiently distinguish the relevant subspaces (i.e., subspaces with clusters) from irrelevant ones.

Part IV

In the fourth part, we filled the gap in the literature of evaluating stream subspace clustering, by presenting a first evaluation framework as well as a first evaluation measure in this area. This part contributes mainly in the *Evaluation and Visualization* step of the KDD process explained in Figure 1.1.

In Chapter 9, the first subspace clustering evaluation framework over data streams, called *Subspace MOA*, is presented. Our open-source framework is based on the MOA stream mining framework, and has three phases. In the online phase, users have the possibility to select one of three most famous summarization techniques to form the microclusters. Upon a user request for a final clustering, the regeneration phase constructs the data objects out of the current microclusters. Then, in the offline phase, one of five subspace clustering algorithms can be selected. The framework is supported with a subspace stream generator, a visualization interface and various subspace clustering evaluation measures.

Chapter 10 contributes a novel external evaluation measure for stream subspace clustering algorithms called *SubCMM*: Subspace Cluster Mapping Measure. SubCMM is able to handle errors caused by emerging, moving, or splitting subspace clusters. Additionally, we extensively compare in this chapter our new measure against state-of-the-art full-space stream clustering evaluation measures. The experimental evaluation, performed using the Subspace MOA framework, depicts the ability of *SubCMM* to reflect the different changes happening in the subspaces of the evolving stream.

Part V

In the final part, we summarize all the contributions given in the different chapters and we give an outlook of the promising future work that can be built over the contributions of this thesis.

Part I

Energy-Efficient Aggregation and Clustering of Sensor Streaming Data

Chapter 2

Energy-Efficient Distributed In-Network Clustering of Sensor Data

[*] Clustering is an established data mining technique for grouping objects based on similarity. For sensor networks, one aims at grouping sensor measurements in groups of similar measurements. As sensor networks have limited resources in terms of available memory and energy, a major task for sensor clustering is the efficient computation within sensor nodes. As a dominating energy consuming task, the communication has to be reduced for a longer battery life. Considering memory, one has to reduce the amount of stored information on each sensor node.

For in-network clustering, k-center-based approaches provide k representatives out of the collected sensor measurements. We propose *EDISKCO*, an **E**nergy-Efficient **D**istributed **I**n-**S**ensor-Network **K**-Center **C**lustering with **O**utliers. Our novel approach is energy-aware and reduces amount of required transmissions while producing high quality clustering results. In thorough experiments on synthetic and real world datasets, we show that our approach outperforms a competing technique in both clustering quality and energy efficiency. Thus, we achieve overall significantly better life times of our sensor networks.

[*]This chapter has been published in the Proceedings of the 3rd International Workshop on Knowledge Discovery from Sensor Data (SensorKDD 2009) held in conjunction with KDD 2009 [HMS09].

2.1 Motivation

Nowadays, sensor networks are deployed in tens of applications from everyday scenarios. In all of these applications, monitoring is the dominating task of WSNs. Collecting useful data from remote sensor nodes is the ultimate goal of researchers and domain people who want to monitor some parameters using the WSN. The collection of all of the sensed data from all of the nodes within the network directly when they are sensed results usually in a perfect information gain about the monitored phenomena. However, not all of the sensed data are *always* interesting "enough" to be sent directly to the gathering station. The resulted excessive energy consumption when sending all sensed data to the gathering station encourages us to consider the previous sentence.

Wireless sensor nodes are spread in many scenarios over mountains or deserts or under the sea, where a continuous energy supply is impossible. In such cases, nodes are powered by batteries with a limited capacity. Moreover, the cost of changing the battery is in most of the cases bigger than getting a completely new sensor node deployed again. Sensor nodes consume energy while sensing, performing internal computations and during the communication of data with other nodes or with the central station. The radio part is the dominating energy consumer. Thus, minimizing the communication times of sensor nodes is mainly targeted when optimizing the energy consumption of a sensor network.

By summarizing the sensed data internally on each sensor node, and then sending only the summaries of these data, one can considerably reduce the updating frequency between the sensor node and the data collecting station. Apparently, this will compromise the information quality of the collected data. To obtain the maximum out of this trade off, some aggregation techniques should be carefully used such that both the information gain and the resources usage time are maximized. Clustering is a data mining task for summarizing data such that similar objects are grouped together while dissimilar ones are separated. In the special case of sensor networks, clustering aims at detection of similar sensor measurements. By detection of k representative measurements in k-center clustering one ensures good clustering quality if each representative is assigned to only very similar measurements. In addition to cluster quality, one aims at an efficient cluster computation. For sensor nodes, resources are limited. Especially, available energy and memory pose restrictions on the clustering algorithms.

The distinguishing features of clustering streaming sensors act as future paths

to research in this area, some of these are: (a) Concerning efficiency issues, sensors are limited in terms of resources, minimizing the consumption of these (mainly energy) is a major requirement to achieve a high lifetime, (b) A compact representation of both the data and the generated model is needed to enable fast and efficient transmission and access from mobile and embedded devices, (c) The final goal is to infer a global clustering structure of all relevant sensors; hence, approximate algorithms should be considered to prevent global data transmission [GGO+08] [GE07].

In this chapter, we propose a novel energy-efficient k-center clustering approach. We propose an incremental algorithm on each sensor node that computes the k representatives. For a compact representation of sensor measurements, we perform an outlier aware clustering by excluding deviating objects from the clustering structure. Our processing enhances clustering quality, specially for sensor measurements where noisy data is gathered. For incremental adaptation of the cluster representatives we use a reclustering phase. However, as each reclustering causes heavy communication costs, our approach aims at reducing such reclusterings, and thus enhances energy efficiency. In addition, we aim at ensuring high clustering quality even with less reclustering operations.

A motivating example for having an energy-aware *k*-center clustering with outliers: Let m distributed sensor nodes over different parts in a certain area in the jungle, and the target is to measure the existence of certain k kinds of wild animals in this area using a combination of sound and vibration measurements. Each sensor node collects readings in its sensing range and performs k clustering over it that reflects its most k existing kinds of animals in its range. A combination of the results will be done in the base station to make a global knowledge about the existence of these kinds in the whole area. The existence of *outlier* animals not expected to be in this area passing over is very normal, which highlights the need to consider outliers. In addition, in such scenarios sensor nodes will be unattached for a long time or even not at all which highlights the need of having energy-aware applications running on it to prolong its battery lifetime. A slightly different scenario to this one is used in [TGC06].

2.1.1 Challenges and Contribution

Many requirements arise when designing an energy-aware in-sensor-network *k*-center stream clustering algorithm that considers outliers; some of these require-

ments are inherited from different areas, the main requirements can be summarized in the following five aspects:

- **Single Passing, Storage awareness:** Due to the limited processing and storage resources in the sensor node, the clustering algorithm must perform only a single pass over the incoming data stream and storage must be independent on n the size of input stream.

- **Minimal Communication:** As the energy consumption of transceiving data between the nodes is usually too big comparing to the computation cost inside the node. The size of data being sent from the sensor nodes to the base station must be minimized.

- **Incremental Clustering:** The algorithm must incrementally cluster the stream data points to detect evolving clusters over the time.

- **High Clustering Quality:** The algorithm must show a good approximation to the optimal clustering by reducing the clustering radius as much as possible, which is the criteria of measuring the k-center clustering quality.

- **Outlier Awareness:** The algorithm should not be sensitive to outliers, nevertheless, it must be able to detect trends in the input stream.

Apparently, not only the two parts of the last aspect are contradicting each other. The second aspect is met when additional in-the-node computations are done to reduce the size of the data to be sent to other node or to the base station, which opposes the first aspect. On the other hand the fulfillment of the third and fourth aspects contradicts again achieving the first one, high-quality incremental clustering needs to store summaries of old points and needs additional processing.

Although many attempts in the literature tried to fulfill the last three aspects, only limited work [COP03, CMZ07, Guh09] has considered the first and the third ones. And to the best of our knowledge, no work has yet been done to consider all the previous aspects together.

Our algorithm considers all above aspects. Motivated by the Global Parallel Guessing algorithm [CMZ07], the key focus of our algorithm is to achieve better clustering quality by using less energy and being resource aware in addition to the consideration of outliers in the input stream.

The remainder of this chapter is organized as follows. Section 2.2 reviews previous work related to our clustering and communicating problem. In Section 2.3 we formulate the related problems. Section 2.4 describes our proposed EDISKCO algorithm in detail. Section 2.5 presents the experimental results. And finally we conclude the chapter in Section 2.6 and discuss some directions for future work.

2.2 Related Work

We will list some previous work done in two strongly related areas: *k*-center clustering approaches and energy-aware routing approaches for sensor networks.

2.2.1 *K*-Center Clustering Algorithms

In the *k*-center clustering problem of a group of points P, we are asked to find k points from P and the smallest radius R such that if disks with a radius of R were placed on those centers then every point from P is covered [Guh09]. The quality of the *k*-center clustering algorithms is measured using the approximation to the optimal clustering. Many clustering solutions have been presented for the *k*-center problem. We will start with reviewing available *offline* approaches which consider that all input data are available in the memory when applying the algorithm, then we will highlight some *online* algorithms which were developed mainly to deal with streaming inputs, we review then solutions which basically targeted the *distribution* of streaming input sources and we end up with solutions which considered the existence of *outliers* in the input stream.

Offline Approaches

These algorithms suggest that all of the n input points are stored in the memory. In an early result, Gonzalez [Gon85] gave the "Furthest Point Algorithm" which gives a 2-approximation to the optimal clustering by making *O(kn)* distance computations. Another 2-approximation approach, called the "Parametric Pruning" was given by Hochbaum and Shmoys [HS85]. The 2-approximation is the best possible solution since it was proved that it is *NP*-hard to find a $(2-\epsilon)$-approximation to the optimal clustering of the *k*-center problem for any $\epsilon > 0$ [FG88].

Online Approaches

Many algorithms were developed to cope up with streaming input, Charikar et. al [CCFM97] introduced the "Doubling Algorithm", a single pass streaming algorithm which guarantees an 8-factor approximation to the optimal clustering and uses *O(k)* space. The main idea is to apply the incremental clustering principle by using a *center-greedy* algorithm which continuously merges two cluster centers, making it possible to maintain new points without increasing the cluster radius. Although they have introduced the principle of incremental clustering with this algorithm, it was not clear how to get the approximation result by incrementally applying this algorithm. Guha [Guh09] presented a $2(1+\epsilon)$ approximation single pass streaming algorithm using only $O(\frac{k}{\epsilon}\log\frac{1}{\epsilon})$ space. The algorithm directly forgets the input data and maintains the stored *summarization* of the input data, this summarization is limited to the available memory space which yields some summarization error. The algorithm is suitable for dealing with stream data by using a memory space independent on n, but it is not suitable for processing-limited machines (like sensor nodes) since it does multiple passing over the stored summarization. Cormode et. al [CMZ07] has formulated the "Parallel Guessing Algorithm" resulting with a $(2+\epsilon)$-approximation to the optimal clustering. This algorithm uses the first points in the input stream to make Δ guesses of R as $(1+\frac{\epsilon}{2}), (1+\frac{\epsilon}{2})^2, (1+\frac{\epsilon}{2})^3, \ldots$ and then scans in parallel this part of the input stream using the different guesses. For each guess, it stops when it returns *k* centers of this input stream using its radius. This will end up by storing $O(\frac{k}{\epsilon}\log\Delta)$ points, where Δ is the maximum number of guesses of the clustering radius that can be done. The smallest guess is then first used for clustering input data. Whenever a new point that is not covered by the recent clustering arrives, the current guess is announced as invalid, and another bigger guessing must be selected. The algorithm is very sensitive to the first k centers selected from received points and some of them might even be outliers, which reduces the clustering quality by using a big guess. The storage is dependent on Δ which can be in reality a big value for the limited storage of sensor nodes. In addition, the parallel nature of the algorithm does suite the limited processing ability of sensor nodes even for small values of Δ.

Distributed Approaches

The idea of having distributed *sites*, each is maintaining a *k*-center clustering algorithm on its local input stream, was originally raised by Cormode et. al [CMZ07]. The idea is to have m remote sites applying the parallel guessing algorithm or an *online* furthest point algorithm on its local data. The site sends its *k*-centers to a central site called *coordinator* which in turn applies another *k*-center clustering algorithm on the $k \times m$ centers. They proved that if the *k*-center clustering algorithm on the site side gives an α-approximation and the one on the coordinator site gives a β-approximation then the resulting k-center clustering algorithm offers a $(\alpha + \beta)$ approximation to the optimal clustering of the whole input data. The suggested distributed algorithm was not mainly targeting sensor networks in the means of energy consumption.

K-Center Clustering with Outliers

The idea of *k*-center clustering with outliers was first presented by Charikar et. al [CKMN01]. They gave an offline algorithm with a 3-approximation which drops z outliers. Our algorithm in contrast drops z far and non-dense outliers online by achieving $(2 + \epsilon)$-approximation. McCutchen and Khuller [MK08] presented another algorithm which gives a $(4+\epsilon)$-approximation using $O(\frac{kz}{\epsilon})$ memory space. The algorithm reads the input in batches of size (kz), stores them, drops all *non-outliers* and then applies an offline k-center clustering algorithm with outliers on them. Our algorithm in contrast performs a single pass over the input points without storing them or performing an offline processing.

2.2.2 Energy-aware Routing Approaches within Wireless Sensor Networks

After collecting their measurements from the physical environment and processing them, sensor nodes have to send these data to one or more base stations. By having the base station(s) within the radio range of each sensor node, the naïve single-hop communication between each node and the base station is possible but not energy-efficient and not reliable because of possible resulting interferences. Additionally, most of the applications do not allow this single-hop deployment, which raised the need for a multi-hop routing path that takes energy efficiency into consideration. This is a pure network communication routing problem which

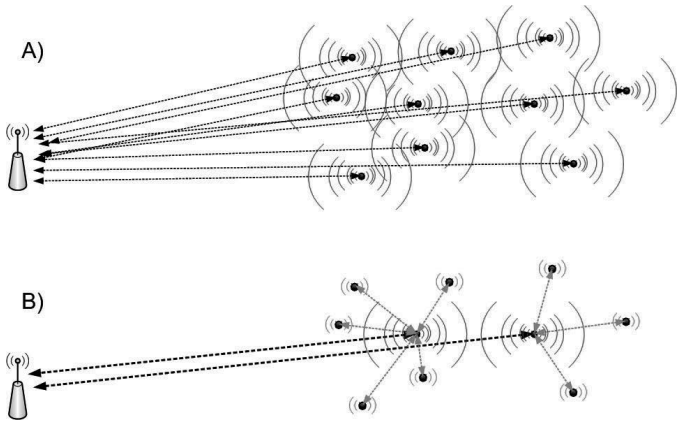


Figure 2.1: An example of a routing protocol: A) All sensor nodes use a big sending power to send their data to the base station. B) Sensor nodes use lower sending power to send data locally to a local node which aggregates and sends them to the base station.

was revisited for energy efficiency concerns. Figure 2.1 illustrates one solution to this problem. Choosing the right underlying routing protocol is particularly important when applying an energy-efficient clustering algorithm. This will guarantee a maximum compatibility for the sake of energy saving. Low-Energy Adaptive Clustering (LEACH) protocol [HCB00] dynamically groups sensor nodes in a small number of clusters. The randomly chosen representatives (cluster heads) locally fuse data from their neighboring and transmit it to the base station, which results in a factor of 8 improvement compared to direct transmissions. In the Hybrid Energy-Efficient Distributed HEED Clustering Approach [OY04] the cluster head selection is mainly based on residual energy and the neighbor proximity of each node. We efficiently apply our algorithm over a networking protocol that extends the lifetime of the wireless sensor network. The protocol efficiently groups local sensor nodes that locally send their data to one of them called coordinator which in turn aggregates these data and sends it to the far base station. The coordinator is iteratively changed depending upon the residual energy which is accurately estimated by our algorithm.

2.3 Problems Formulation

We formulate in this section the related problems to our algorithm.

2.3.1 The *K*-center Problem

Given a set *P* of *n* points $P = \{p_1, \ldots, p_n\}$, a distance function $d(p_a, p_b) \geq 0$ satisfying the triangle inequality and an integer $k < n$, a *k*-center set is $C \subseteq P$ such that $|C| = k$.
The *k*-center problem is to find a *k*-center set C that minimizes the maximum distance of each point $p \in P$ to its nearest center in C; i.e., a set that minimizes the quantity $max_{p \in P} min_{c \in C} d(p, c)$. The well-known *k*-median clustering problem is the the minsum variant of this problem where we seek to minimize $\sum_{p \in P} min_{c \in C} d(p, c)$ [Guh09].

2.3.2 The Incremental Clustering Problem

Let $S = \{c_1, c_2, \ldots, c_k, R\}$ be a *current* solution of a *k*-center clustering algorithm **A** applied on n input points that are arriving to the algorithm one by one in a sequence of updates. **A** is an incremental clustering algorithm if it can always maintain a valid solution over the flow of stream. In other words, whenever a new point arrives to the algorithm it should either be assigned to one of the clusters indicating the validity of current clustering, or it does not fit in any of the current clusters then the current S must be changed into another solution S' such that this new point is assigned to some cluster in the new solution S'. S' can differ from S by the centers, radius or by both of them.

2.3.3 The Distributed Clustering

In distributed clustering we track each sensor node data locally, process it and then combine the results in a central node or a *coordinator*. The target is to minimize the communication and share the resources. We define the **distributed clustering problem.** Let $1, 2, \ldots, d$ be distributed sites, each site i applies a clustering algorithm A_i on his stream input of data X_i and produces a solution S_i. It is required to perform a global clustering of all X_i; $i = 1 \ldots d$ input streams distributed over the sites. One efficient solution to do that is to have a central

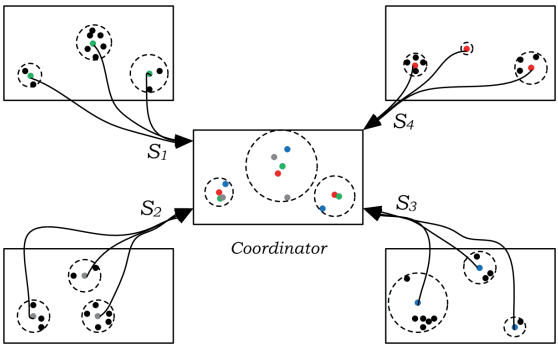


Figure 2.2: An example of a distributed *k*-center clustering for k=3 clusters of data coming from d=4 sites, the coordinator applies another *k*-center clustering over the $k \times d$ centers sent from the sites.

site which collects the union $\bigcup_{i=1}^{d} S_i$ and answers the querying or monitoring requests of the whole input streams. It is also possible that a further clustering algorithm B at the coordinator to be applied on $\bigcup_{i=1}^{d} S_i$. In the distributed *k*-center clustering problem we will consider in this chapter, the solution of A_i at each site i is $S_i = \{c_{i1}, c_{i2}, \ldots, c_{ik}, R_i\}$, and on the coordinator side, we perform another *k*-center clustering algorithm over the whole *k* centers coming from the whole d sites, i.e., $\bigcup_{i=1}^{d}\{c_{i1}, c_{i2}, \ldots, c_{ik}\}$. When applying incremental clustering algorithms, continuous updates with the new solutions must be sent from the sites to the coordinator. Figure 2.2 illustrates an example where the coordinator applies another *k*-center clustering over the $k \times d$ centers sent from the sites.

2.3.4 The Problem of *K*-center Clustering with Outliers

Taking the existence of some outliers in the input stream into account is a natural consideration. Since data is often noisy and this can dramatically affect the quality of the solution if not taken into account. This particularly applies to the *k*-center objective function which is extremely sensitive to the existence of points far from cluster centers [MK08]. This sensitivity has two effects: (a) On the clustering quality which appears in Figure 2.2 for example in site 3 where the cluster on the left has a bigger clustering radius (worse clustering quality) because of not considering one point too far from the center as an outlier. And (b) On the energy consumption in the distributed model since outliers cause current solu-

tions to be invalid and need to be updated with the coordinator. Formally in the ***k*-center clustering problem with outliers** we group m out of the n input points into the *k* clusters by dropping $z = n - m$ points, the target is to minimize the clustering radius. The decision of labeling those z points as *outliers* is done when they are farther than R from the current *k* centers and the number of neighboring *non-clustered* points is not "enough" to establish a new cluster.

2.3.5 The Energy Cost Model

Considering sensor nodes that contain sensing, processing and radio parts. The total energy consumption E of a sensor node can be defined as:

$$E = E_{comp} + E_{transc} + E_{sensing} + E_{sleep} \tag{2.1}$$

where:

- $E_{comp} = E_{proc} + E_{acces}$: the energy consumption of the computations, divided into E_{proc} the consumption of processing done by the microprocessor and E_{acces} the energy cost of accessing an external memory to load or store the data.
- $E_{transc} = E_t + E_r$: the energy consumption of transmitting E_t and receiving E_r done by the radio.
- $E_{sensing}$: the energy consumption of the sensors.
- E_{sleep}: the energy consumption when the microprocessor is in the sleep mode.

The energy consumption of each part depends both on the time and the current draw of this part when it is active. By considering both E_{comp} and E_{transc}:

$$E_{comp} = E_{proc} + E_{acces} = V \cdot I_c \cdot t_c + V \cdot I_a \cdot t_a \tag{2.2}$$

$$E_{transc} = E_t + E_r = ss_t \cdot V \cdot I_t \cdot t_t + V \cdot I_r \cdot t_r \tag{2.3}$$

Where:
V: the working voltage of the sensor node
I_c: the current draw of the microprocessor in the active mode
t_c: the time in which the microprocessor is running

I_a: the current draw of accessing the external memory
t_a: the time taken to access the external memory
ss_t: the transmitting signal strength of the radio
I_t: the current draw of the radio when transmitting
t_t: the time when the radio is transmitting
I_r: the current draw of the radio when sending
t_r: the time when the radio is sending

Let c_t, c_r be the size of the data to be transmitted or received, and f_{transc} be the radio speed, then: $t_t = f_{transc} \cdot c_t$ and similarly: $t_r = f_{transc} \cdot c_r$, this makes Equation 2.3 look like:

$$E_{transc} = E_t + E_r = f_{transc} \cdot V \left(ss_t \cdot I_t \cdot c_t + I_r \cdot c_r \right) \tag{2.4}$$

Usually in sensor nodes, the current draw of the microprocessor unit I_c is too small compared to I_a, I_t and Ir, in addition, the microprocessor speed is also too big compared to f_{transc} or the external memory bus speed. All that together makes E_{transc} and E_{acces} too big compared to E_{comp} in the sensor node. This highlights the importance of reducing both E_{transc} and E_{acces} in an in-sensor-node energy-aware clustering algorithm. E_{acces} is reduced by limiting the need to access the external memory as much as possible, a target which can be achieved by making as less as possible passes over the input data. And from the cost model in Equation 2.4, it can be seen that E_{transc} can be reduced by reducing c_t and c_r the size of the data to be sent or received, while an energy-aware routing protocol tries to reduce ss_t.

2.4 The EDISKCO Algorithm

In the section we present the Energy-efficient Distributed In-Sensor *K*-center Clustering algorithm with Outliers EDISKCO. Each local sensor node receives its input stream through its sensors and produces a *k*-center clustering solution to it by considering the existence of outliers and sends this solution to the coordinator. The coordinator performs another clustering algorithm to the solutions coming from the sites. Therefore, the input streams are processed at each node locally, and a global clustering of all sensors data is performed globally in the coordinator . We will describe the algorithm on the node side, the coordinator side and

on the server side. We introduce a special heap structure for storing a $k+z$ members, each member c_j; $j = 1 \ldots (k+z)$ in this heap represents a cluster, where: $c_j.center$ represents the center of this cluster and $c_j.count$ represents the number of members inside this cluster (density). The members in this heap are arranged in a descending order according to $c_j.count$. The member on top of the heap where $(j = 1)$ represents the cluster with the highest density.

Algorithm 2.1: ***insert(h,p)***

In	*A heap h of current clusters and an input point p from the stream*
Out	*0:if OK, j: if p is a new cluster in position j or err:if there is no place to add a new cluster p*

```
1:  for j = 1 to size(h) do
2:    if d(p, c_j.center) ≤ R then
3:      c_j.count+ = 1
4:      maintain(h)
5:      return 0
6:    end if
7:  end for
8:  if size(h) < (k + z) then
9:    c_j.center = p
10:   c_j.count = 1
11:   return j
12: else
13:   return err
14: end if
```

We define the following functions on h:

- ***maintain(h)***: applied after each change in $c_j.count$; $j = 1 \ldots k+z$ such that if $j > q$ then $count_q \geq count_j$ for all $q = 1 \ldots k+z$ and $j = 1 \ldots k+z$.
- ***size(h)***: returns the number of the members in h which can be any value between 0 and $k+z$.
- ***get(h,j)***: returns the member j from the heap.
- ***delete(h,j)***: deletes the member j from the heap and directly maintains the heap.
- ***insert(h,p)***: inserts an input point p from the stream in h, (see Algorithm 2.1). It scans the members of h beginning with the high dense ones. When

a cluster is found where p is not further than R from its center, this cluster's counter is incremented by one, and p is forgotten. If p was further than R from all available cluster centers and there was less than $k + z$ members in h, then a new cluster is established with p as its center. Otherwise an error is returned for not having a place to add p.

2.4.1 On The Node Local Side

Each node i receives an input stream $X(i)$, runs the NodeSideEDISKCO algorithm (see Algorithm 2.2) and sends updates to the coordinator with *k* center outlier-aware clustering representation of $X(i)$ in addition to the corresponding radius R_i. Whenever an input point p is received, the algorithm applies the $insert(p, h)$ function. If p fits in one of the available clusters then its member counter is increased, if not then it is necessary to establish a new cluster containing p as its center. We limit the number of clusters to be established to $(k + z)$: k most dense clusters for representing the solution and z lowest dense centers for representing the outliers. The main target of our algorithm is to reduce the communication with the coordinator by excluding outliers from final solution, so only if this new established cluster falls in the first k most dense clusters then an update is sent to the coordinator containing new_center signal and the point p. If we need to establish a new cluster for p and we already have $(k+z)$ clusters, then the current clustering is no longer valid and the solution needs to be maintained. The node i does this by increasing the current radius. In order to select the best new radius size, the node i sends a $radius\_increase$ request message to the coordinator. The coordinator in turn replies to i with an acknowledgment message with the biggest clustering radius it has noticed from all nodes (R_{global}). Node i buffers its input points from $X(i)$ till it receives the acknowledgment from the coordinator. When it receives it, the node i selects $max\{R_i(1 + \frac{\epsilon}{2}), R_{global}\}$ as the next radius, and starts scanning the available clusters in h beginning with the highest density cluster by comparing the distance between their centers with the new R_i. If a center $j + 1$ is less than the R_i away from the center j where $j = 1 \ldots k + z$, then the cluster member $j+1$ is deleted, the counter of the cluster j is increased, and j is compared with the new member in $j + 1$ again. If the member in $j + 1$ is farther than R_i from j, then $j + 1$ is kept and the same comparison happens between $j + 1$ and $j + 2$. Finally, we take the most dense $l < k$ resulting clusters available and delete the others.

Algorithm 2.2: NodeSideEDISKCO$(X_i, k, z, o, l, \epsilon)$

In *An input stream X_i, num of clusters k, outlier clusters num z, maximum outliers num o, num of most dense clusters l and step size ϵ*

Out *update the coordinator with the new opened clusters centers and radii*

$X_i(t) \rightarrow p, c_1.center = p, c_1.count = 1, R = R_{min}$
while there is input stream $X_i(t)$ **do**
  $X_i(t) \rightarrow p$
  $fit = insert(p, h)$
  if $next\_coordinator$ signal from server **then**
    Save current local centers in C_{local}
    Switch to *CoordinatorSideEDISKCO*
  end if
  if $fit \neq err$ **and** $0 < fit \leq k$ **then**
    Send the new center $get(h, fit)$ with a new_center signal to the coordinator.
  end if
  if $fit == err$ **or** $\sum_{j=k+1}^{size(h)} c_j.count > o$ **then**
    Send $radius\_increase$ request to coordinator
    while there is no reply from coordinator **do**
      Buffer incoming input points from X_i
    end while
    $R \leftarrow max\{R(1 + \frac{\epsilon}{2}), R_{global}\}$
    while $j < size(h)$ **do**
      if $d(c_j.center, c_{j+1}.center) \leq R$ **then**
        $c_j.counter++$, $delete(h, j + 1)$
      else
        $j++$
      end if
    end while
    Keep only the most l dense centers in h.
    Run this algorithm on buffered points.
    $C_i = \{c_1.center, c_2.center, \ldots, c_{\hat{k}}.center\}$
    Send C_i, R_i as an update to the coordinator
  end if
end while

The last scanning procedure guarantees that we are keeping the effect of the most l dense clusters which appeared in history solutions. On the other hand, taking only l clusters will leave a space for the establishing new clusters if there was a new trend in the input stream. This is very essential for incremental clustering algorithm on the one side, on the other side this will reduce the number of $radius\_increase$ requests and consequently the energy consumption of the node. In the initialization phase of the NodeSideEDISKCO algorithm (not shown in Algorithm 2.2 for readability), the node increases the radius without contacting the coordinator until $\sum_{j=1}^{k} c_j.count \geq n$, where n is the minimum number of points that we want to represent using the k center. Only in this case the current solution and radius are valid and can be sent to the coordinator. The algorithm continues the comparison on the buffered input points from $X(i)$ and sends the resulting centers with the used radius to the coordinator. The decision of sending a $radius\_increase$ request to the coordinator is made also if the total number of points which considered to be in the outliers became more than o. Having more than o outliers close to each other in z or less clusters means that the decision of considering those points as outliers is no longer valid and needs to be changed by a new clustering. Because the coordinator is performing more communication and computation tasks, our algorithm is also aware of iteratively changing the coordinator in order not to have one node exhausted and died. The decision of choosing the node i as the next coordinator is made by the server (as we will see) and based on the residual energy of all nodes. In this case, the server sends a $next\_coordinator$ message to i. The node i saves its local solution C_{local} and switches to the coordinator algorithm then a new *phase* begins.

2.4.2 On The Coordinator Side

The coordinator receives the local solutions C_i, radii R_i and $radius\_increase$ requests from the nodes and performs the CoordinatorSideEDISKCO algorithm (see Algorithm 2.3). When a new phase begins, the new coordinator receives the global solutions C_{global} and the global radius R_{global} from the previous coordinator. Then the new coordinator announces himself to the other nodes as the new coordinator. The coordinator continuously performs the Furthest Point algorithm [Gon85] on the solutions C_i arriving from the sites. It starts by applying it on its own local solution C_{local}, and then on C_i. The coordinator saves always the largest radius used by a node as the R_{global} and whenever any node i wants to

increase its clustering radius, the coordinator replies an acknowledgment with R_{global}. The coordinator also adds the new centers $c_{ij}.centers$ arriving from the nodes to the current solution.

Algorithm 2.3: CoordinatorSideEDISKCO($C_i, R_i, c_{ij}.center$)

In *solutions C_i, R_i, new opened cluster centers and cluster_increase requests from node i; $i = 1 \dots m$*

Out *send ack and R_{global} to nodes, maintain C_{global}, R_{global} and send them to next coordinator*

```
1:  if this is not the first coordinator then
2:      Receive C_global, R_global from last coordinator
3:  end if
4:  Broadcast I_am_coordinator signal to all nodes
5:  C_global ← FurthestPoint(C_global, C_local)
6:  if radius_increase_i do
7:      Send to node i an ack with R_global
8:      Receive C_i, R_i from node i
9:      C_global ← FurthestPoint(C_global, C_i)
10:     R_global ← max{R_global, R_i}
11: end if
12: if new_center_i do
13:     C_global ← FurthestPoint(C_global, {c_ij.center})
14: end if
15: if consumption_update received from server then
16:     Send {numCenters_i, numRequests_i} for all i = 1 ... m
        nodes during this phase to the server
17: end if
18: if chg_coordinator received from server then
19:     Send C_global, R_global to next coordinator
20:     Switch to NodeSideEDISKCO using R_global
21: end if
```

The coordinator keeps a special space for saving summary about the energy consumption of each node i. The total number of centers that were sent from a node i and the total number of $radius\_increase$ requests sent from a node i during this phase are saved under $numCenters_i$ and $numRequests_i$ respectively. These are important for the server to calculate the total energy consumption of each node, including the coordinator, during this phase. The server sends from time to time $consumption\_update$ requests to the coordinator which in turn replies to them. According to the residual energy of each node and the coordina-

tor, the server makes a decision of changing the current coordinator by sending $chg\_coordinator$ message to it, with the *id* of the next coordinator. In case of having a huge number of nodes, the server can group the nodes into different groups each covered by one coordinator, this is to avoid a possible collapse of a single coordinator due to the heavy messages traffic or the too big data to be saved in memory. One idea to do this is by considering the, already known by the server, spatial information of the node in making these groups and then considering the residual energy when deciding the next coordinator of each group. On the local side of each group, since each coordinator is transceiving data using minimal sending strength, then this will guarantee that a node will be covered by only one coordinator. It might happen that a node receives more than one $I\_am\_coordinator$ signal, then it decides to be covered by the coordinator with the strongest signal, which is basically its group coordinator. Before the end of the next phase, the coordinator sends C_{global} and R_{global} to the next coordinator, and switches to the node algorithm.

2.4.3 On The Server Side

The far server is responsible for selecting the coordinator for the next phase according to the residual energy in each node. All of the nodes start with the same level of energy. The server starts by randomly selecting one of them as the coordinator of this phase. And then from time to time, the server collects the $numCenters_i$ and $numRequests_i$ of each node from the coordinator. The server uses these information and the Equation 2.1 to estimate the energy consumption of each node as well as the coordinator. All nodes are sensing samples with the same frequency, so the server can estimate the total number of points taken in this phase $numPoints_i$.

The coordinator is collecting all C_i requests from the nodes and replying with the acknowledgments and R_{global} to each request. On the other hand it is answering the queries of the far server using its full sending strength and performing more complicated multipass *k*-center clustering algorithm over C_i's. Thus it is consuming much more energy comparing to the normal nodes. If we keep using the same coordinator, it will die after a while, and we will lose the connection to the other *still alive* nodes.

The target is to change the coordinator to the node that contains the most residual energy, such that all of the nodes in the network will die as close as

possible to each other. This will let us make maximum use of the whole network lifetime.

The server assumes the following worst cases: (a) all the $numCenters_i$ points needed k comparisons until they were placed in a new cluster and then sent, (b) all the $(numPoints_i - numCenters_i)$ points needed $(k+z)$ comparisons until they were saved and (c) Only l centers were sent at once to the coordinator after sending $raduis\_increase$ to it, while the other $(numCenters_i - l \times numRequests_i)$ were sent one by one to the coordinator during this phase. Although these conservative assumptions might be a little bit far from the real case, the server can continue using the last coordinator even if it was supposed to die as long as it is *really* still alive.

2.4.4 Lower Bounds of EDISKCO

EDISKCO algorithm was mainly motivated by the Global Parallel Guessing algorithm (Global-PG) which has an altogether $(4+\epsilon)$-approximation to the optimal clustering quality to the global optimal solution [CMZ07]. On the coordinator side, both PG and EDISKCO algorithms are using the Furthest Point algorithm which has a 2-approximation to the optimal clustering (see [CMZ07] for proof). After sending the first solution to the coordinator, our algorithm incrementally maintains a *k*-center solution for at least n input points. The solution stays valid unless we have more than z clusters of outliers or more than o input points assigned as outliers. Following the same way used in proving the lower bound of the parallel guessing algorithm in [CMZ07] one can proof that the NodeSideEDISKCO finds a $(2+\epsilon)$-approximation to the optimal *k*-center clustering of n input points. EDISKCO does not exclude the non-outlier points from the final solution, it simply delays the decision of increasing the radius until one is sure that those points are not really outliers. EDISKCO then either considers them in the final solution if they have formed a dense cluster, or reclusters by taking all of them into consideration. By applying the Furthest point algorithm on the solutions on coordinator side which has a 2-approximation, the EDISKCO algorithm finds an altogether $(2+\epsilon)+2 = (4+\epsilon)$-approximation of the optimal global clustering solution, by saving only $(k+z)$ points and making a maximum of $(k+z)$ comparisons for each input point. Due to its insensitivity to the first input points and outliers in addition to its incremental nature compared to Local-PG, we will show in the next section that our algorithm has a better clustering

Dataset	Size	Nodes	EDISKCO	Global-PG
RW	42000	19	**573382.4**	587015.5
i9-Sensor	40589	19	**547270.2**	558121.8
Physio	24000	12	**205970.6**	214380.8

Table 2.1: Total energy consumption of EDISKCO and Global-PG for each dataset in Joule

quality and energy consumption than the Global-PG.

2.5 Experimental Evaluation

In order to evaluate the performance of EDISKCO, we performed extensive experiments on both synthetic and real data. For comparison, we have chosen the Global-PG as a single-pass distributed *k*-center clustering algorithm on the node side which applies also the furthest point algorithm on the coordinator side. In order to have fair results, we have implemented our suggested node-coordinator-server model on the Global-PG. We have implemented simulations of both algorithms in Java.

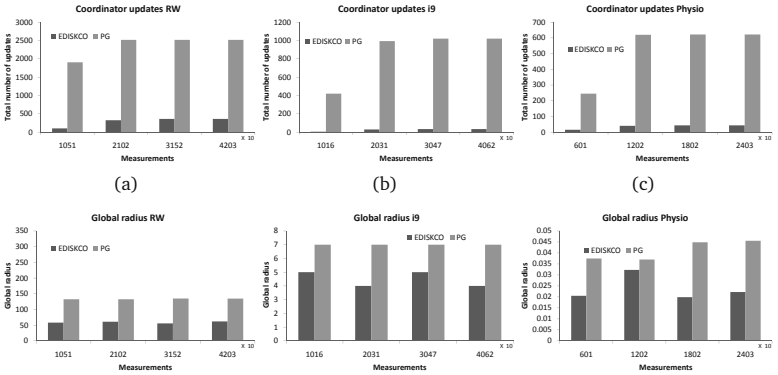
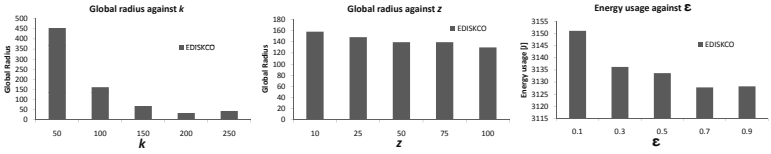


Figure 2.3: Total number of requests (a)-(c), R_{global} (d)-(f) of the: RW, i9-Sensor and Physio datasets

We have chosen one synthetic dataset and two real world datasets and we give here a small description of each.



(a) (b) (c)

Figure 2.4: Parameter sensitivity (a)-(b) in RW, (c) in i9-Sensor

Synthetic Dataset: RandomWalk (RW)[†] A synthetic dataset based on the random walk model. The increments between two consecutive values are independent and identically distributed. Each increment: t_{i+1} is produced by randomly adding or subtracting from t_i a uniformly random value from the interval [1,10] . We generated 19 different datasets each for one node, each containing 42,000 measures. Subsequently, to produce a natural outliers effect, we replaced randomly selected values (4.5% of the dataset size) with noise values, uniformly at random in the interval $[min, max]$ of the dataset.

Real Dataset: I9 Sensor Dataset[‡] We have collected a real data from a sensor network. We deployed 19 TelosB motes in our department area. All motes were programmed to collect temperature samples each 30 seconds and send them directly to a sink connected to a computer. The data was collected for more than 14 days between the 10th and the 23rd of April 2009 and forming 40,589 measures of each node. The minimum difference between raw measures is 1. Nodes were not always able to communicate perfectly with the sink due to collisions or loss of signal, this appeared in 2.9% of the total data. Instead of each measure that did not reach the sink, we introduced a noise data. In a different way of adding outliers to that of RW, a uniformly at random value from the interval $[0.1 \times (max - min), 0.25 \times (max - min)]$ was selected and then uniformly at a random either added to max or deducted from min, the resulting value was inserted instead of the lost measure.

Real Dataset: Physiological Sensor Dataset [Phy] This data was presented in ICML 2004 as a challenge for information extraction from streaming sensor

[†]The RW is available under http://dme.rwth-aachen.de/EDISKCO
[‡]The I9 dataset is available under http://dme.rwth-aachen.de/EDISKCO

data. The training dataset consists of approximately 10,000 hours of this data containing: userID, gender, sessionID, sessionTime, annotation, characteristic 1, characteristic and sensor[1..9]. We have extracted the first 24,000 readings of sensor2 by userID. We have chosen the data of 12 different userIDs with the same gender, each representing a node. We did not add outliers to this dataset as they are naturally exist in such datasets.

2.5.1 Evaluation Criteria and Methodology

Three criteria were chosen to evaluate our algorithm, namely:

Clustering Quality: We have selected the global clustering radius as a measure to evaluate the quality of EDISKCO with Global-PG. In *k*-center clustering, better clustering uses smaller radius to cover all of the input points.

Energy Consumption: Two measures were selected to evaluate the energy efficiency of EDISKCO: (a) Total number of reclustering events, which tells about the total number of costly requests sent from all the nodes to the coordinator for increasing the current clustering radius. (b) Total energy consumption of the whole network in Joule based on the detailed cost model suggested in Equation 2.1 and the datasheets of TelosB mote, TI MSP430 microcontroller and the CC2420 radio chip in addition to the TinyOS 2.0.2 operating system installed on the motes.

Parameter Sensitivity: We wanted to see the effect of varying ϵ, k and z parameters on the previous measures of our algorithm.

2.5.2 Experimental Setup and Results

For performing the **clustering quality and the energy consumption evaluations**, we have selected for our algorithm for all datasets: $k = 200$, $z = \frac{k}{4}$, $\epsilon = 0.5$, the number of the most dense clusters to be sent to the coordinator after each cluster increase: $l = \frac{k}{4}$, the maximum allowed number of input points after which the node can send a solution to the coordinator: $n = 400$ and the maximum allowed number of outliers in total: $o = 10\%n$. For the Global-PG we have selected the number of points collected at the beginning to perform the parallel guessing equals to our $n = 400$ and also $\epsilon = 0.5$. We performed the evaluations on a 3.00 Ghz core Duo, 4 GB RAM machine.

For both algorithms, we need to set the initial radius value such that: $R_{min}(P) = min_{p,q \in P, p \neq q} d(p, q)$ for each dataset. This is usually possible since

sensor nodes are dedicated for certain measurements and that we have experts in the field who can decide this value. Mostly this value is already determined by the precision of the sensors. For the I9 sensor dataset R_{min} equals to 1, but for the physiological and RW datasets this was too small comparing to R_{max}. The direct selection of the R_{min} in those two datasets was extremely affecting running time of the Global-PG algorithm due to the enormous number of guesses to be done in this case. Even the desktop machine we used for evaluation was hanging. This is one of the main drawbacks of Global-PG when applied on resource limited sensor nodes. In contrast, we have not observed that effect with our algorithm. For the experiments we have selected R_{min} equals to 1 for the RW and equals to 0.0001 in the physiological dataset.

For performing the **parameter sensitivity** evaluations, we selected $k = 100$ other parameters are same as above. For both algorithms we applied our suggested routing protocol for changing the coordinator, the server was lazily querying the current coordinator for $numCenter_i, numRequests_i$. For the three datasets and the two algorithms, Table 2.1 represents the total energy consumption in Joules, while Figure 2.3 depicts the accumulated total number of reclustering events and the current global radius with respect to the flow of input stream.

Figure 2.3(a) shows that EDISKCO sends 85% less updates to the coordinator than Global-PG, and Figure 2.3(d) shows that EDISKCO achieves 55% better clustering quality of the RW data. Because all of the guesses are done on first received input stream, Global-PG needs to update the coordinator whenever it receives a new point which does not fit in the current clustering. This results in a heavy node-coordinator communication from one side, and a bigger clustering radius to include the new points on the other side. This effect appears even in the dataset which does not contain introduced outliers like Physio dataset (Figures 2.3(c) and 2.3(f)), where EDISKCO achieves 93% less number of updates and 14% to 58% better clustering quality. Another reason of this low updates number of EDISKCO is its scanning method when inserting a new point, which starts by scanning dense clusters that are more likely to contain additional points. In consequence, EDISKCO consumes less total energy than Global-PG.

Table 2.1 shows that EDISKCO saves 2.32% compared to Global-PG when applied on RW dataset and 2%, 3.9% when applied on I9-Sensor and Physio datasets respectively. The saving is considerable for extending the lifetime of the whole network. In most of the updates, Local-PG sends only the one new center. Our cost model suggests a direct relation between E_{transc} and the size of data to

be sent. This explains the differences between improvement percentages of Table 2.1 and Figure 2.3. In reality, heavy communication consumes more energy due the higher possibility of collisions and signal loss.

In Table 2.1, Global-PG consumes additional energy in the Physio dataset due to the huge number of guesses done at first when considering $R_{min} = 0.0001$. Figures 2.4(a) and 2.4(b) present the effect of changing k parameter in the interval [0, 250] and the z parameter in the interval [0, k] on R_{global} in the RW dataset.

Figure 2.4(c) depicts the effect of changing the accuracy parameter ϵ on the single node usage in the i9-Sensor dataset. It shows that the less the clustering accuracy, the less the energy consumption, a natural relation which highlights the significance of our approach by achieving better clustering quality in combination with longer lifetimes.

2.6 Conclusion

In this work we presented our novel energy-efficient *k*-center clustering solution. As a single-pass algorithm we developed an incremental processing which is aware of outliers in the data. We enhanced the clustering quality by excluding these outlying objects from the clustering. Furthermore, we reduced the cost of intensive reclustering operations and achieved lower energy consumption. For the limited energy resources of sensor networks, our energy-efficient computation induces longer lifetimes of the network. In thorough experiments, we presented the high clustering accuracy and low energy consumption of our approach. Furthermore, our algorithm is also aware of limited memory resources in recent sensor nodes.

Chapter 3

Weighted Outlier-Aware *K*-Center Clustering of Sensor Data

* Collecting data from sensor nodes is the ultimate goal of Wireless Sensor Networks. This is performed by transmitting the sensed measurements to some data collecting station. In sensor nodes, radio communication is the dominating consumer of the energy resources which are usually limited. Summarizing the sensed data internally on sensor nodes and sending only the summaries will considerably save energy.

In this chapter we propose a novel weighted, resource-aware **Sen**sor data k-center **Clu**stering algorithm called: *SenClu*. Our algorithm immediately detects new trends in the drifting sensor data stream and follows them. *SenClu* powerfully uses a light-weighted decaying technique that gives lower influence to old data. As sensor data are usually noisy, our algorithm is also outlier-aware. In thorough experiments on drifting synthetic and real world datasets, we show that *SenClu* outperforms two state-of-the-art algorithms by producing higher clustering quality and following trends in the stream while consuming nearly the same amount of energy.

3.1 Motivation

In this chapter, we see another challenge in addition to those mentioned in the Section 2.1.1 when designing an energy-aware in-sensor-network *k*-center stream

*This chapter has been published in the proceedings of the 4th International Conference on Networked Digital Technologies (NDT 2012) [HS12b] and the Journal of Digital Information Management (JDIM) 2012 [HS12a].

clustering algorithm that considers outliers. Namely: the algorithm must incrementally cluster the stream data points to detect evolving clusters over the time, while forgetting outdated data. We propose a novel energy-efficient k-center clustering approach: *SenClu* that incrementally groups the data locally on each sensor node and computes its k representatives. The algorithm gives more importance to newly received data for a faster adaptation to the evolving trends in the stream and more information gain of recent data on the collecting station. SenClu improves the clustering quality by using a novel light-weighted decaying technique to give less importance to old sensed values. It uses also a smart merging technique for the decayed clusters. The clusters with the least weight represent decaying data and are smartly treated in SenClu as outliers. This gives a space for grouping the new emerging clusters, and thus following the current trend in the input streams. We sustain the powerful features of EDISKCO [HMS09] but show that SenClu produces considerably better clustering results than EDISKCO [HMS09] and another state-of-the art competing algorithm while consuming nearly the same amount of energy.

The remainder of this chapter is organized as follows: Section 3.2 will introduce some preliminaries. Section 3.3 describes our proposed SenClu algorithm in detail. And finally Section 3.4 presents the experimental results. And finally we conclude the chapter in Section 3.5.

3.2 Preliminaries

3.2.1 Weighted *K*-Center Clustering

A weighted *k*-center clustering algorithm **A** uses the following structure to save information about a clustering $\mathcal{C}$: $\{c_1, w_1, t_{u1}, c_2, w_2, t_{u2} \ldots, c_k, w_k, t_{uk}, R\}$. Where w_i is the weight of cluster i, c_i is its center and t_{ui} is the last time when the cluster i was updated by a point from the stream input. Let t_{now} be the current time, then the weight of cluster i is calculated as follows:

1. $[w_i = 2^{-\lambda \cdot (t_{now} - t_{ui})}]$ if the cluster i was not updated by the current stream input point (at time t_{now}).

2. $[w_i = 2^{-\lambda \cdot (t_{now} - t_{ui})} + 1]$ if the cluster i was updated by the current stream input point (at time t_{now}).

Where $1 \leq \lambda \leq 0$ represents the decaying factor. Larger values of λ result in a faster decaying of old members of the current cluster, while smaller values represent more contribution of old members in the calculation of the weight of the current cluster.

3.2.2 The Problem of Weighted *K*-center Clustering with Outliers

Sensor data are usually noisy. If a clustering algorithm is not outlier-aware the results will become extremely sensitive to the noise. This sensitivity has two effects: (a) On the clustering quality which appears in Figure 2.2 for example in site 3 where the cluster on the left has a bigger clustering radius (worse clustering quality) because of not considering one point too far from the center as an outlier. And (b) On the energy consumption in the distributed model since outliers cause current solutions to be invalid and result in additional updates with the coordinator. Formally, (cf. Section 2.3.3) in the weighted ***k*-center clustering problem with outliers** we group m out of the n input points into the k clusters by dropping maximally $z = n - m$ points. The decision of labeling those z or less points as *outliers* is done according to their small weights compared to other clusters. Noisy data form less dense clusters that receive updates less frequently. The weight of such clusters will soon decrease.

3.3 The SenClu Algorithm

In this section we present SenClu. Each local sensor node receives its input stream through its sensors, produces a weighted k-center clustering solution to it by considering the existence of outliers and then sends this solution to the coordinator. The coordinator performs another clustering algorithm to the solutions coming from the sites. A server part of the algorithm manages iteratively assigning coordinators and receiving data from them. Therefore, similar to EDISKCO [HMS09], the input streams are processed at each node locally, and a global clustering of all sensors data is performed globally in the coordinator.

SenClu uses for a heap structure h for storing maximally $k + z$ weighted clusters. Each member c_j; $j = 1 \dots (k + z)$ in this heap represents a cluster, where: $c_j.center$ represents the center of this cluster, $c_j.weight$: the weight and $c_j.up\_time$: the last time when c_j was updated. The members in this heap are

Algorithm 3.1: ***insert(h,p)***

In *A heap h of current clusters and an input point p from the stream*

Out *0:if OK, j: if p is a new cluster in position j or err:if there is no place to add new cluster p*

1: **for** $j = 1$ to $size(h)$ **do**
2: **if** $d(p, c_j.center) \leq R$ **then** *// decay all clusters then and*
3: *//increase the weight of j by one using maintain(h)*
4: $maintain(h)$;
5: **return** 0;
6: **end if**
7: **end for**
8: **if** $size(h) < (k + z)$ **then** *// there is still a place in the // heap to insert p*
9: $c_j.center = p$;
10: $c_j.weight = 1$;
11: *maintain(h)*;
12: **return** j
13: **if** $c_{k+z}.weight \leq w_{min}$ **then** *// there is an old cluster, // replace it*
14: $delete(h, k + z)$;
15: *//maintain the weights using maintain(h)*
16: $maintain(h)$;
17: **return** (the insertion position)
18: **return** err */* we have to recluster */*

arranged in a descending order according to $c_j.weight$. The top k members represent the clusters, while the rest which could be maximally z represent the outlier clusters. Arranging the clusters according to the weights needs to be done only once after each reclustering. Once the members are arranged, only the updated cluster needs to be rearranged such that it is in the correct place of the list. All non-updated clusters decay with the same factor together. We define the following functions on h:

- ***maintain(h)***: applied after each reclustering or birth of a new cluster such that for all $1 \leq j, q \leq k + z$ we have $q > j$ only if $c_q.weight \geq c_j.weight$. Whenever a point is inserted in a cluster, *maintain(h)* simply performs a decaying step for all clusters weight: $c_i.weight = 2^{-\lambda} \times c_i.weight$ for all $1 \leq i \leq k + z$, and then in the next step increases only the weight of the cluster where the input point was inserted by 1. The decaying step will leave the order of the heap correct, after the increasing step, one scan step is needed to insert the updated cluster in its correct place in the arranged

list. The previous step is performed to avoid the complicated mathematics associated with calculating: $w_i = 2^{-\lambda \cdot (t_{now} - t_{ui})} + 1$ in general, which most sensor node processors can not afford.

- ***size(h)***: returns the number of the members in h which can be any value between 0 and $k + z$.
- ***get(h,j)***: returns the member j from the heap.
- ***delete(h,j)***: deletes the member j from the heap and directly maintains the heap.
- ***insert(h,p)***: inserts an input point p from the stream in h, (see Algorithm 3.1). It scans the members of h beginning with the high-weighted ones. When a cluster is found where p is not further than R from its center, all the clusters are aged by $2^{-\lambda}$, and only the found cluster's weight is incremented by 1, and p is forgotten. If p was further than R from all available cluster centers and there were less than $k + z$ members in h, then a new cluster is established with p as its center. Otherwise check if the least weighted cluster $k + z$ has less weight than w_{min} (the minimum weight), if yes, delete it and insert the new point in a new cluster and return its position. Otherwise, an error is returned for not having a place to add p.

3.3.1 On The Node Local Side

Each node i receives an input stream $X(i)$, runs the SenCluNode algorithm (see Algorithm 3.2) and sends updates to the coordinator with *k* center outlier-and-weight-aware clustering representation of $X(i)$ in addition to the corresponding radius R_i. Please mind that during the initialization phase (not shown in Algorithm 3.2 for readability), the node increases the radius without sending updates to the coordinator until n input points are received. Then the running phase of SenCluNode starts. SenCluNode is explained in details in Algorithm 3.2.

3.3.2 On The Coordinator Side

The coordinator side algorithm is explained in SenCluCoordinator (cf. Algorithm 3.3). Lines 15-20 explain the communication messages between the server and

Algorithm 3.2: SenCluNode$(X_i, k, z, \lambda, w_{min}, \epsilon)$

In *An input stream X_i, num of clusters k, num of outlier clusters z, decaying factor λ, minimum weight w_{min} and step size ϵ*

Out *update the coordinator with the new opened clusters' centers and radii*

1: $X_i(t) \rightarrow p, c_1.center = p, c_1.weight = 1, R = R_{min}$
2: **while** there is input stream $X_i(t)$ **do**
3: $X_i(t) \rightarrow p$
4: $fit = insert(p, h)$
5: **if** $next\_coordinator$ signal from server **then** *//current node // becomes coordinator*
6: Save current local centers in C_{local} *// in order not to lose // its local solutions*
7: Switch to *SenCluCoordinator*
8: **end if**
9: **if** $fit \neq err$ **and** $0 < fit \leq k$ **then**
10: Send the new center $get(h, fit)$ with a new_center signal to the coordinator.
11: **end if**
12: **if** $fit == err$ **then** *// we need to recluster by increasing the radius*
13: Send $radius\_increase$ request to coordinator
14: **while** there is no reply from coordinator **do**
15: Buffer incoming input points from X_i
16: **end while**
17: $R \leftarrow max\{R(1 + \frac{\epsilon}{2}), R_{global}\}$ *// get the biggest possible next R*
18: **while** $j < size(h)$ **do** *// merge the overlapping clusters*
19: $q = j + 1$;
20: **while** $q < size(h)$ **do**
21: search for the closest j and q to each other;
22: increase the weight of the higher-weighted of them by 1;
23: delete the lower-weighted one;
24: $q++$;
25: **end while**
26: $j++$;
27: **end while**
28: $maintain(h)$;
29: Keep only the the clusters whose weight is at least w_{min} in h;
30: Run this algorithm on buffered points;
31: $C_i = \{c_1.center, c_2.center, \ldots, c_{\hat{k}}.center\}$; *// get the // heaviest $\hat{k} \leq k$*
32: Send C_i, R_i as an update to the coordinator;
33: **end if** *// end of the reclustering phase*
34: **end while**

Algorithm 3.3: SenCluCoordinator($C_i, R_i, c_{ij}.center$)

In *solutions C_i, R_i, new opened cluster centers, cluster_increase requests from node i*

Out *send ack and R_{global} to nodes, maintain C_{global}, R_{global} and send them to next coordinator*

1: **if** this is **not** the first coordinator **then**
2: Receive C_{global}, R_{global} from last coordinator; *// undertake the // solutions*
3: **end if**
4: Broadcast $I\_am\_coordinator$ signal to all nodes;
5: $C_{global} \leftarrow FurthestPoint(C_{global}, C_{local})$; *// as in [Gon85]*
6: **if** $radius\_increase_i$ **do** *// received: radius_increase request from // node i*
7: Send to node i an ack with R_{global};
8: Receive C_i, R_i from node i; *// new solution from i w.r.t. the // new value of R*
9: $C_{global} \leftarrow FurthestPoint(C_{global}, C_i)$; *// update C_{global} with the // new solution*
10: $R_{global} \leftarrow max\{R_{global}, R_i\}$; *// update R_{global} with the new solution*
11: **end if**
12: **if** new_center_i **do**// *coordinator received: new centers on node i*
13: $C_{global} \leftarrow FurthestPoint(C_{global}, \{c_{ij}.center\})$
14: **end if**
15: **if** $consumption\_update$ received from server **then**
16: Send $\{numCenters_i, numRequests_i\}$ for all $i = 1 \ldots m$ nodes during this phase to server
17: **end if**
18: **if** $chg\_coordinator$ received from server **then**
19: Send C_{global}, R_{global} to next coordinator
20: Switch to *SenCluNode* using R_{global}
21: **end if**

the coordinator for managing the selection of the next coordinator according to the residual energy that each node still possesses. The coordinator keeps a special space for saving summary about the energy consumption of each node i. The total number of centers that were sent from a node i and the total number of $radius\_increase$ requests sent from a node i during this phase are saved under $numCenters_i$ and $numRequests_i$ respectively. These are important for the server to calculate the total energy consumption of each node including the coordinator during this phase. The server sends from time to time $consumption\_update$ requests to the coordinator which in turn replies to them. The server uses an energy model similar to the one in [HMS09]. According to the residual energy of each node and the coordinator, the server makes a decision of changing the current coordinator by sending $chg\_coordinator$ message to it, with the *id* of the next coordinator.

3.4 Experimental Evaluation

We have evaluated the performance of SenClu using extensive experiments on both synthetic and real data. As competitors, we have chosen two state-of-the-art algorithms: EDISKCO [HMS09] and Global-PG [CMZ07] (we refer to it as PG in the experiments). Both competitors perform a single-pass distributed *k*-center clustering algorithm on the node side and the furthest point algorithm on the coordinator side. In order to have fair results, we have implemented our suggested node-coordinator-server model also on the Global-PG. We have implemented simulations of the three algorithms in Java. We have used the same three datasets used mentioned in Section 2.5. One synthetic dataset: RW and two real world datasets: I9-Sensor and Physio.

We have used the following three criteria to evaluate SenClu w.r.t. EDISKCO and the PG algorithm:

Silhouette Coefficient: We use this measure to evaluate the clustering quality on the nodes side. It reflects how appropriate the mapping of data objects to clusters is. It subtracts the average distance of objects to their representative from the average distance of objects to their second closest cluster and then divides the results over the bigger average. When calculating the average of these values for all objects in all clusters, the final value will range from -1 to +1. Where -1 will reflect the worst clustering and +1 the perfect one. For the streaming case, we have used a sliding window over the stream input and then performed the

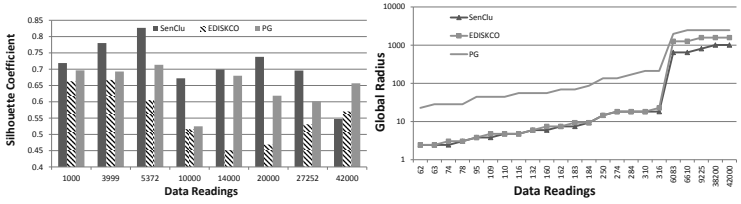
calculation of the Silhouette coefficient at the end of each window for all the objects within it.

Global Clustering Radius: Another measure to reflect the clustering quality, this time on the coordinator side. In *k*-center clustering, better clustering uses smaller radius to cover all of the input points.

Energy Consumption: Evaluated through the average energy consumption of the one sensor node in the network in Joule based on the detailed cost model suggested in [HMS09] and the datasheets of TelosB mote, TI MSP430 microcontroller and the CC2420 radio chip in addition to the TinyOS 2.0.2 operating system installed on the motes.

3.4.1 Experimental Setup and Results

For all experiments, we selected the parameters for SenClu and EDISKCO for all datasets as: $k = 15$, $z = \frac{k}{4} = 4$, $\epsilon = 0.5$. For SenClu only, we selected: $w_{min} = 0.5$, λ as: (0.005 in RW dataset), (0.018 in I9 Sensor Dataset) and (0.01 in Phyosiological Dataset). For EDISKCO only: the number of the most dense clusters to be sent to the coordinator after each cluster increase: $l = \frac{k}{4} = 4$, the maximum allowed number of input points after which the node can send a solution to the coordinator: $n = 100$, and the maximum allowed number of outliers in total: $o = 10\%n$. For the Global-PG we have selected the number of points collected at the beginning to perform the parallel guessing equals to our $n = 400$ and also $\epsilon = 0.5$. We performed the evaluations on a 3.00 Ghz core Duo, 4 GB RAM machine. For all algorithms we set the initial radius as $R_{min}(P) = min_{p,q \in P, p \neq q} d(p, q)$ for each dataset P separately. Figure 3.1(a) shows



(a) (b)

Figure 3.1: The clustering quality using the Random Walk Synthetic Dataset over different parts of the input stream data. (a) Silhouette Coefficient, (b) R_{global}.

that SenClu achieves considerably higher Silhouette coefficient values than both EDISKCO and PG over almost all of the data stream of the RW dataset. This high clustering quality of SenClu is due to its novel weighing technique that allows new emerging trends to influence the clustering result, and thus to be grouped in the correct cluster. Also on the clustering performed on the coordinator side, Figure 3.1(b) shows that SenClu has always smaller global radius than EDISKCO which constitutes a better clustering. PG has considerably worse performance than SenClu and EDISKCO on the coordinator side (mind the logarithmic scaling in Figure 3.1(a)). Also in Figure 3.2(a), we can see that SenClu has a better clustering performance on the node side than both competitors over the whole data measurements of the I9 Sensor Dataset. Figure 3.2(b) shows again that the decaying nature and the smart merging technique that SenClu has, result in a smaller radius on the coordinator side and thus better overall clustering results than both PG and EDISKCO. Figure 3.3(a) shows on another real dataset (Physiological Sensor Dataset) that SenClu always has a better clustering quality than both competitors on the node side. Because PG is more sensitive to noise than SenClu and EDISKCO, it is performing considerably worse than the others on this relatively noisy dataset. Figure 3.3(b) is showing that on the node side, SenClu is having most of the time the same global radius as EDISKCO. Only for a short time, SenClu is having a bigger radius than EDISKCO. Table 3.1 shows

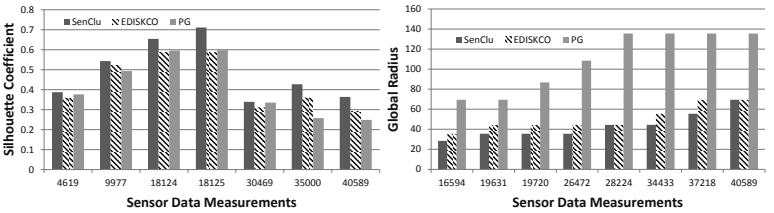


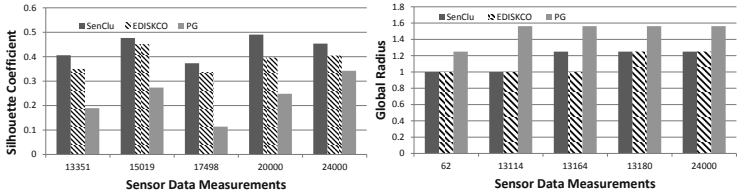
Figure 3.2: The clustering quality using the I9 Real Sensor Dataset over different parts of the input measurements. (a) Silhouette Coefficient, (b) R_{global}.

that SenClu consumes a bit more energy on average than both EDISKCO and PG when applied on RW dataset. This can be explained by the random nature of the Random Walk dataset that results in different new trends in the data, that SenClu tries to follow. This results in multiple updates to the coordinator of new

Dataset	Size	Nodes	SenClu	EDISKCO	PG
RW	42000	19	29805.7	**29776.4**	29792.11
i9-Sensor	40589	19	28770.2	**28768.38**	28792.5
Physio	24000	12	**17074.3**	17074.4	17078.9

Table 3.1: Average energy consumption in Joule of a single node in the network by the end of each dataset when using SenClu, EDISKCO and PG.

created clusters. This is not the case for EDISKCO and PG where a very lazy update of newly emerging clusters saves some energy while extremely affects the clustering quality (cf. Figures 3.1(a) and 3.1(b)). This effect does not appear when using natural real datasets. We can see from Table 3.1 that on I9 Sensor Dataset, SenClu consumes less than two Joules more than EDISKCO, and absorbs considerably less energy than PG. When using the Physiological Sensor Dataset, SenClu consumes less energy than both competitors.



(a) (b)

Figure 3.3: The clustering quality using the Real Physiological Sensor Dataset over different parts of the input stream data. (a) Silhouette Coefficient, (b) R_{global}.

3.5 Conclusion

In this chapter, we present our novel energy-efficient weighted k-center clustering solution. We presented our algorithm: *SenClu* as a single-pass algorithm that immediately detects new trends in the drifting sensor data stream and follows them. The light-weighted decaying technique which we used to enhance the clustering quality, gives lower influence to old data. As sensor data are usually noisy, *SenClu* is also outlier-aware. In thorough experiments on drifting synthetic and real world datasets, we showed that our approach clearly outperforms two

state-of-the-art algorithms by producing higher clustering quality and following trends in the stream while consuming nearly the same amount of energy.

Chapter 4

Energy-Efficient Self-Adaptive Clustering of Sensor Nodes

* Physical clustering of nodes in sensor networks aims at grouping together sensor nodes according to some similarity criteria like neighborhood. Out of each group, one selected node will be the group representative for forwarding the data collected by its group. This considerably reduces the total energy consumption, as only representatives need to communicate with distant data sink. In data mining, one is interested in constructing these physical clusters according to similar measurements of sensor nodes. Previous data mining approaches for physical clustering concentrated on the similarity over all dimensions of measurements.

In this chapter we introduce the *ECLUN* algorithm: an **E**nergy-aware method for physical **CLU**stering of sensor **N**odes based on both spatial and attributional similarities. The approach uses a novel method for constructing physical clusters according to similarities over some dimensions of the measured data. In an unsupervised way, the introduced method maintains physical clusters and detects outliers. Through extensive experiments on synthetic and real world datasets, it is shown that ECLUN outperforms a competing state-of-the-art technique in both the amount of consumed energy and the effectiveness of detecting changes in the sensor network. Thus, an overall significantly better life times of sensor networks is achieved, while still following changes of observed phenomena.

*This chapter has been published in the Proceedings of the 4th International Workshop on Knowledge Discovery From Sensor Data (SensorKDD 2010) held in conjunction with KDD 2010 [HMS+10].

4.1 Motivation

The communication process is the dominating energy consumer in sensor networks, particularly when this is happening over long distances. Previous two chapters (Chapter 2 and Chapter 3) covered the aggregation of the sensor measurements within each node of the network, for the sake of a longer network life. This chapter is covering the grouping of different sensor nodes according to their similar readings. Sensor nodes need to use their full sending power to forward their sensed data to distant sink, while they can use less power when communicating locally with each other. Considering the energy limited resources in sensor networks, this motivated a lot of research on the physical clustering of sensor nodes. The idea is to divide sensor nodes into groups according to some criteria, and then select one node from each of these groups to serve as a group representative. The main task of the group representative is forwarding the readings of sensor nodes from its group to this distant sink. As nodes need to communicate within the group (the cluster) using less energy, this considerably reduces the total consumed energy in the whole network.

Data mining approaches contributed to this problem mainly in two parts: the criteria used for clustering and the process of selecting representatives. The similarity of sensed measurements and spatial characteristics was used as a grouping measure. Thus, inside each cluster, the node with the most similar readings to the measurements of all nodes inside that cluster is selected as a cluster representative.

In both cases, the selection methodology is based on the similarity between all attributes of clustered nodes. Todays sensor nodes are collecting increasingly many number of dimensions for each sensor node. The similarity measures should cope with the increasing dimensionality of sensed data. In such data, distances grow more and more alike. The full data space is thus sparse and each node will be alone in its physical cluster as no global similarity between the measurements of different nodes can be observed. We are tackling this point in this chapter by introducing a novel method for performing physical clustering based on the similarity over some of the sensed attributes using subspace clustering. We show that this method produces improvements in energy consumption even for low dimensional data.

In addition to the importance of saving energy, we designed our method to cope with the change detection. Detecting novelty in input stream is an important

feature that has to be considered when designing any data knowledge technique in sensor networks. For example, it is an essential point in evaluating learning algorithms in drifting environments [GGO+08, Agg13].

4.1.1 Our Contribution

The following aspects are our main contributions that we included in this work:

- **Reducing the communication burden**
  In our approach, nodes do not continuously communicate with the representative. Communication is established only when a state change is detected in the monitored phenomena. By the careful construction of clusters, this communication is further reduced by using the similarity to representative readings.

- **Subspace physical clustering**
  Our novel method for building clusters according to the relevant attributes results in more consistent clusters, and helps for maintaining the clusters with less effort.

- **Outlier-aware change detection**
  We present a simple but effective method for detecting outliers in the input stream performed by each node, and another one performed by the representative to detect deviating nodes in its cluster. We show that our approach by applying this method, is still capable of detecting changes in input stream.

- **Uniform utilization of energy resources in sensor network**
  We suggest further optimization methods to our approach to uniformly distribute the usage of energy between the nodes. We cope with the cases of single-node clusters, and changing representatives according to residual energy. This results in a longer lifetime of the whole sensor network as nodes die close to each other.

Figure 4.1 simplifies the idea of ECLUN. A smart and adaptive representative selection process is continuously performed not only by considering spatial similarities, but also similarities over some dimensions of measured data. Only the representatives (red) will send their measurements to the sink.

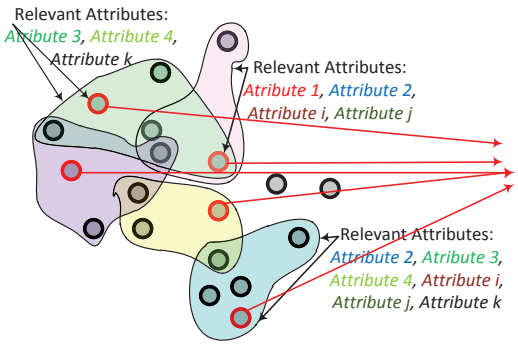


Figure 4.1: An example how ECLUN performs the grouping of nodes and the selection of representatives additionally according to similarity w.r.t. some attributes.

The remainder of this chapter is organized as follows. Section 4.2 mainly reviews the literature related to the physical clustering problem. Section 4.3 introduces some formulations and definitions used in our approach. Section 4.4 describes in detail our algorithm. Section 4.5 presents the experimental results. We conclude the chapter and suggest future work in Section 4.6.

4.2 Related Work

In this section we list briefly the related work to our physical clustering problem.

Traditional offline clustering algorithms e.g. [EKSX96], [ABKS99],[ZRL96] can not cope with the streaming and distributed nature of sensor nodes.

Although some **distributed versions of clustering algorithms** were established like SDBDC [JKP04], DFEKM [JGA06], they are still dealing with offline data and can not simply adapt to perform online distributed clustering.

Many algorithms were developed to deal with the **online distributed** clustering of data. The Distributed Grid Clustering algorithm [RGL08] is an example of an online 2-layer distributed clustering of sensor data. ELink [MS06] and the Distributed Single-Pass Incremental algorithm DSIC [YG08] are two examples on time series clustering of sensor nodes. None of these algorithms considered the possibility of having clusters hidden in subsets of the attributes.

Subspace clustering has been proposed, for today's applications with increas-

ing number of given dimensions. Subspace clustering detects clusters in arbitrary projections by automatically determining a set of relevant dimensions for each cluster [PHL04, KKZ09]. Thus, one is able to detect objects as part of various clusters in different subspaces. Recent research has seen a number of approaches using different definitions of what constitutes a subspace cluster [AGGR98, KKK04]. As summarized in a recent evaluation study [MAG+09b], their common problem is that the generated output is typically huge. In recent subspace clustering algorithms, we have focused on tackling redundancy [AKMS07, AKMS08, MAG+09b]. In contrast, projected clustering assigns each object to a single projection [AWY+99, MSE06]. This strict partitioning of the data into projected clusters can be regarded as extreme redundancy elimination. Projected clustering results in a manageable number of clusters, but is not able to detect overlapping clusters. Both subspace clustering and projected clustering have their focus on offline data outside sensor networks. In contrast, we aim at combining clustering in subspace projections [AKMS07, AWY+99] with physical clustering for sensor networks [HMS09].

SERENE [BC06] is a framework for SElecting REpresentatives in a sensor NEtwork. It uses clustering techniques to select the subset of nodes that can best represent the rest of sensors in the network. In order to reduce communication, rather than directly querying all network nodes, only the representative sensors are queried. In this way, the overall energy consumption in sensor network is reduced and consequently sensor network lifetime is extended. To select an appropriate set of representative sensors, SERENE performs the analysis of historical readings of sensor nodes, in order to find out the correlations both in space and time dimensions among sensors and sensor readings. Sensors may be physically correlated. Sensor readings may be correlated in time. Physically correlated sensors with correlated readings are assigned to the same cluster. Then each cluster performs further analysis in order to select the sensors with the highest representation quality. The last two steps of this process are the same with the steps of clustering process in our algorithm. Similar to our algorithm, this technique uses density-based clustering algorithm, DBSCAN [EKSX96]. Nevertheless, different from our algorithm, in SERENE approach the first stage of clustering process is analysis of historical data for detecting correlations among nodes and sensor readings. Due to restrictions of energy, computational and memory capacity in sensor nodes, this analysis can not be performed by the nodes themselves. Continuous storing of historical data for all nodes that are spatially correlated,

in order to analyze correlation of their readings, requires more memory capacity than a sensor node possesses. Processing of all the analysis over measurements of sensors to find out correlations needs high computation resources as well. Moreover, this process requires exchange of attribute measurements between all the nodes that are spatially correlated. This is followed by a high energy consumption in nodes, due to frequent communication and data exchange with more than one node in their clusters. Due to all these restrictions, in this approach sensor nodes can not be self organized into clusters. As a result, this technique is suitable only for those scenarios where nodes operate in a supervised way. Another difficult part of this technique is related with the maintenance of SERENE platform. With passing of time, the readings of sensor nodes change, consequently the same set of sensors may not be anymore correlated with each other, or a new correlation may appear among some other nodes. This change requires a reorganization of nodes in clusters. Reclustering process is followed by additional communication among nodes for updating historical data. This will increase the communication burden and the size of transmitted data will be significantly high. More analysis should be performed over data, meaning more resources will be consumed for computation purposes.
All the above mentioned reasons make this approach expensive in terms of energy and not easy to maintain in cases of continuous clustering applications.

In [SBF+07] a Data-Driven Processing Technique in Sensor Networks was suggested. The goal of this technique is to provide continuous data without continuous reporting, but with checks against the actual data. To achieve this goal, this approach introduces temporal and spatio-temporal suppression schemes, which use the in-network monitoring to reduce the communication rate to the central server. Based on these schemes, data is routed over a chain architecture. At the end of this chain, the nodes that are most near to central server send the aggregate change of the data to it.

Snapshot Queries [Kot05] is another approach that introduces a platform for energy-efficient data collection in sensor networks. By selecting a small set of representative nodes, this approach provides responses to user queries and reduces the energy consumption in the network. In order to select its representative, each sensor node in this approach builds a data model for capturing the distribution of measurement values of its neighbors for each attribute. After a node decides which of its neighbors it can represent, it broadcasts its list of candidate cluster members to all its neighbors. Each node selects, as its representative, its neigh-

bor that can represent it and that additionally has the longest list of candidate cluster members. This is again expensive as all messages are broadcasted and not directed to specific nodes, which might result in repeated broadcasting in case of message loss. Maintaining this model is very expensive in terms of energy, as all nodes need to exchange all historical readings among each other. In our algorithm, each sensor node maintains a small cache of past measurements of itself for each attribute. And the control messages exchanged among nodes during the initialization phase are directed to specified nodes. As the closest state-of-the-art to our approach, we evaluate our algorithm by comparing it to Snapshot Queries [Kot05].

4.3 Problem Formulation

In this section we formally define the related problems to our algorithm.

4.3.1 The Representatives Selection Problem

Given a set SN of n sensor nodes $SN = \{sn_1,\ sn_2, ..., sn_n\}$ each measuring a set of attributes $\{a_1, a_2, ..., a_k\}$, an Euclidean distance function $d(sn_a, sn_b) \geq 0$; $\{a, b\} \subset \{1, 2, ..., n\}$ and a real number $\varepsilon > 0$. The problem of selecting representative nodes in SN is to find a subset $R = \{r_1, r_2, ..., r_m\} \subseteq SN$; $m \leq n$ each $r_i \in R$ is representing a set of nodes $D_i = \{sn_{i1}, sn_{i2}, ..., sn_{il}\}$, $D_i \subseteq SN$ and $\forall sn \in D_i$: $d(r_i, sn) \leq \varepsilon$ such that the measurements sensed by all members of D_i are best represented by the measurements of r_i.

Definition 4.1 *(Physical cluster of nodes) A physical cluster C of sensor nodes is a set D_i with a maximum number of $MaxNds > 0$ nodes represented by the representative r_i such that $\forall sn \in D_i$:*

$$\sqrt{(x_{r_i} - x_{sn})^2 + (y_{r_i} - y_{sn})^2 + (z_{r_i} - z_{sn})^2} \leq \varepsilon$$

where ε is the radius of C.

Definition 4.2 *(Spatial and non-spatial attributes) Each node $sn \in SN$ is defined in each time stamp t by a set of attributes $\{a_{1t}, a_{2t}, ..., a_{kt}, x_{sn}, y_{sn}, z_{sn}\}$ where $\{a_{1t}, a_{2t}, ..., a_{kt}\}$ is a set of non-spatial attributes which represent the measurements of sn at time stamp t and $\{x_{sn}, y_{sn}, z_{sn}\}$ are the spatial attributes of sn.*

Definition 4.3 *(Relevant attributes) Let* $\{\mu_1(t), \mu_2(t), \cdots \mu_k(t)\}$ *and* $\{\sigma_1(t), \sigma_2(t), \cdots \sigma_k(t)\}$ *be respectively the mean values and the standard deviations of the non-spatial attributes of* l *readings of sensor node* sn_a *at time stamp* t*, a non-spatial attribute* a_m*, where* $1 \leq m \leq k$*, is called a relevant attribute between two nodes* sn_a *and* sn_b *at the time* t *if* $X_{mt}(sn_b) \in [\mu_m(t) - 2\sigma_m(t), \mu_m(t) + 2\sigma_m(t)]$ *where* $X_{mt}(sn_b)$ *is the sensor* sn_b *reading of attribute* m *at time stamp* t*.*

4.3.2 The Problem of the ECLUN Algorithm

Given a set SN of n sensor nodes with a set of attributes deployed in an environment for monitoring physical phenomena and a base station to collect these measurements, the general problem of ECLUN algorithm is to decrease the total amount of consumed energy in SN by grouping the nodes of SN into physical clusters C_i where $1 \leq i \leq n$ each represented by a representative r_i with some relevant attributes a_j where $1 \leq j \leq k$ and then sending to the base station either the readings of the relevant attributes of r_i to represent the readings of all members of C_i or the summary of the readings of all members of C_i. The target is to continuously update the base station by all important changes in the sensed phenomena.

4.4 The ECLUN Algorithm

In this section, we describe in details our approach. We differentiate between two phases of the algorithm. The initialization phase where physical clusters are constructed in an unsupervised way, and the running phase when these clusters are maintained and updates are sent to the representative and the server.

4.4.1 The Initialization Phase

Algorithm 4.1 gives an overview of this phase. We will next describe each of these steps in details.

Caching of Initial Data

Each node senses the first l measurements for each attribute and stores them in the cache of data history. These l measurements will be exchanged between nodes, and thus will decide the initial physical clustering of nodes. Therefore,

Algorithm 4.1: Initialization Phase of ECLUN

1: *Caching of initial data*
2: *Detection of geographical neighbors*
3: *Setting relevant attributes*
4: *Estimation of representation quality for each node*
5: *Selection of local representatives*
6: *Load balancing among representatives*

outlier readings must completely be excluded in this phase. We assume that attribute measurements for each sensor are normally distributed. Therefor, nodes during this phase continuously calculate the mean and standard deviation values of their measurements for each attribute. If any new reading falls out the corresponding confidence interval $[\mu - 2\sigma, \mu + 2\sigma]$, it is suspected to be an outlier, and stored in the suspected list with a maximum length of s. If the suspected list was filled within the previous s time stamps, then its readings are considered in the main list, otherwise it is excluded completely from both lists.

Detection of Geographical Neighbors

Every node detects its *geographical neighbors* GN by running spatial queries with radius ε. This is done by broadcasting its ID and spatial attributes and the mean values of non-spatial attributes $< ID_n, \mu_1, \mu_2, \cdots, \mu_k, x_{sn}, y_{sn}, z_{sn} >$ within a radius ε, where $\mu_1, \mu_2, \cdots, \mu_k$ are the mean values of the initial l readings of sn for each non-spatial attribute. Thus, every node becomes aware of the geographical coordinates as well as the initial readings of its neighbors, these data are stored in a list GN in each node.

Setting Relevant Attributes

In this step, each node decides the relevant attributes between it and each node in its GN list. According to Definition 4.3, the node uses the non-spatial readings received in the previous step and the statistics of its own previous l measurements to decide the relevant attributes. The threshold $Min\_Rel\_Attr \leq k$; k is the number of non-spatial attributes, decides the minimum number of relevant attributes between two nodes when one wants to represent the other. Each node excludes from the list of GN, all neighboring nodes with less than Min_Rel_Attr relevant attributes to it. The rest nodes are stored in the *candidate cluster member* CCM list.

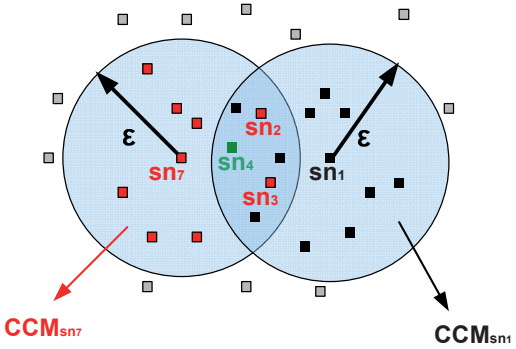


Figure 4.2: Candidate Cluster Members

In Figure 4.2, although nodes sn_2 and sn_3 are part of GN of node sn_1, they do not belong to its *candidate cluster members* CCM. Apparently, there are less than Min_Rel_Attr relative attributes between node sn_1 and each of sn_2 and sn_3, so they are both not in the CCM of sn_1.

Estimation of Representation Quality for Each Node

In this step of algorithm, each node analyzes how effective it is in representing its CCM nodes in the network.

Definition 4.4 *(Representation quality)*
The representation quality $RepQ$ of node sn when representing its CCM nodes is defined as:

$$RepQ(sn) = (1-\alpha)\frac{\sum_{sn_i \in CCM(sn)}(\varepsilon - d(sn, sn_i))}{\varepsilon \times \mid CCM(sn) \mid} + \alpha\frac{RE_{sn}}{IE_{sn}}$$

Where ε is the maximum radius of the possible cluster that might be represented by sn, $d(sn, sn_i)$ is the distance between sn and any of its CCM, α is a coefficient for weighting the energy, IE_{sn} and $RE_{sn}$are the initial and the residual energy of sn respectively.

According to Definition 4.4, $RepQ$ is greater when the members of CCM are forming a compact cluster around sn. Closer nodes mean less consumed energy and much more similar measurements. Additionally, the residual energy

is an important factor, as the possibility will be less later that sn gets soon out of energy. The bigger the value of α, the more the importance of the residual energy factor when selecting representatives gets.

Selection of Local Representatives

Each node decides whether it will be itself a local representative or it will be represented by any other similar node in its neighborhood. To take this decision, nodes refer to the representation quality parameter.

Each node sn_i broadcasts its $RepQ$ value to every node sn_j that belongs to its CCM. Every node $sn \in SN$ stores the list of *candidate local representatives* CLR, together with the $RepQ$ values received by them, and includes itself in this list. The list is ranked in a decreasing order according to $RepQ$. One of the following will happen:

1. If the current node has its own $RepQ$ value at the top of this list, it announces itself as a representative.

2. If two nodes have the same $RepQ$ value, then the closer node is selected as a representative for the current node

3. Otherwise, node sn is represented by the node which is having the $RepQ$ in its CLR

After this step, every node either has chosen only one node as its representative, or is a representative itself. Since representatives announce themselves, each node collects the IDs of representative nodes in its neighborhood and it stores them in an internal list called *neighbor local representatives* NLR.

As we saw when building CCM, we had $Min\_Rel\_Attr \leq k$; k is the number of non-spatial attributes, we adopt this idea from the subspace clustering area [AKMS07, AWY+99]. For many given attributes, one can hardly find two sensor nodes that can have similar measurements over all attributes. This fact can result with a huge number of single-node clusters. To avoid this, we relax the representation criteria in such a way that the representative needs only to have some relevant attributes with its represented nodes. Algorithm 4.2 gives a description of the process of selecting the representative according to the relevant attributes and updating the server with relevant and non-relevant attributes by each node.

Algorithm 4.2: Selecting representatives per attributes

1: **if** *this_attribute* is a *relevant_attribute* **then**
2: Let it be represented by the local representative rep_a which is sharing the highest number of relevant attributes
3: **else if** (other representative rep_b can represent *this_attribute*) **then**
4: Some attributes are represented by rep_a others by rep_b
5: **else**
6: Let *this_attribute* be forwarded to server by rep_a
7: **end if**

Load Balancing Among Representatives

To provide a uniform utilization of energy resources in sensor network, we set a threshold $MaxNds$ for the *maximum number of nodes* that can be represented by one representative. According to that, representatives decide to exclude from its cluster the most distant cluster members. The excluded node then tries to join the nearest representative in its NLR list.

At the end of the initialization phase, physical clusters C are established.

4.4.2 The Running Phase

The algorithm initiates the communication process only when a state change is detected. Nodes communicate with their representatives only when they detect a state change in the attribute measurements of the event they are monitoring. Similarly, representatives send data to the server only if they detect a state change in the statistics of the measurements collected from all the nodes of their clusters. We have then two possible communication paths: node-representative and representative-server.

Node-Local Representative Communication

Each node sn compares the current measurement values $X_{jt_i}(sn)$ on non-spatial attribute $j = (1 \cdots m)$ sensed at t_i with the mean value $\mu_j(t_{i-1})$ of the l previous measurements values of the corresponding attribute. If $|X_{jt_i}(sn) - \mu_j(t_{i-1})| \leq \delta_j$ where δ_j; $j = (1 \cdots m)$ are the measurements thresholds for attribute j, then a change in the measurements is detected and an update of X_{jt_i} should be sent to

the corresponding representative. Otherwise, no data is sent to the representative and old measurements sent previously to the representative by sn are used.

Local Representative-Server Communication

During each time stamp in the running phase, the representative executes Algorithm 4.3. After this, and at the same time stamp, the representative maintains C. It checks whether $X_{jt_i}(sn)$ that it has received from sn falls inside the confidence interval $[\mu_{jC}(t_{i-1}) - 2\sigma_{jC}(t_{i-1}), \mu_{jC}(t_{i-1}) + 2\sigma_{jC}(t_{i-1})]$ for each relevant attribute in C or not. If this was not the case, then sn is temporarily excluded from the t_i statistics. Its readings are saved in a list with a maximum length s. If s was filled within the previous s time stamps with readings of sn, then the representative requests sn to join another physical cluster, and forwards its s readings together with the ID of sn to the server. And sn, in turn, searches for a neighboring representative in its NLR list and continues from step 5 in Algorithm 4.1. The only exception here will be that Min_Rel_Attr threshold does not apply, as nodes try to minimize the number of nodes representing them.

Algorithm 4.3: Representative running phase

```
1: while updates are received from nodes at time t_i do
2:   if any attribute is missed then
3:     use t_{i-1} values
4:   for each relevant attribute j do
5:     μ_jC(t_i) = 1/|C| Σ_{sn∈C}
6:     if |μ_jC(t_i) − μ_jC(t_{i-1})| ≤ ψ_j then
7:       update the server with μ_jC(t_i) and σ_jC(t_i)
8:   end for
9: end while
```

4.4.3 Energy-Aware Optimizations

We suggest further optimizations for the sake of energy efficiency in our algorithm.

Delegation of Representative Authority

The energy of local representatives decreases rapidly much more than the energy of other nodes in the network.

If a representative runs out of energy, all of its cluster nodes should recluster.This is considerably energy consuming. Furthermore, losing the representative node will cause a big lack of information about the monitored phenomena delivered by the complete cluster. We suggest a uniform utilization of energy sources in the network by applying a technique of delegation representative authority.

Each local representative is aware of its residual energy. At the time it notices that its energy capacity is decreased under a certain threshold (for instance: 50% of its initial energy as it started to represent this cluster), local representative requests the residual energy values of nodes in C. The authority of representing the cluster is delegated to the node with the highest residual energy including current representative. If none of the cluster nodes has more residual energy than current representative, then it continues being the representative of its cluster and performs later the same check again.

Optimization in Case of Single Node Cluster

In such a scenario, sensor node has to communicate with distant server for updating only its measurements. To avoid this, each node that is alone in its cluster sends lazily *"join requests"* to its neighbor representatives. Each neighbor representative then checks whether the attribute measurements are relevant to its cluster. Accordingly it might join that cluster or keep representing itself. In case of more acknowledgments, it selects the nearest neighbor representatives. Receiving no-acknowledgment means that the node is selecting different data than its neighbors and will keep representing itself. This might mean that either this node is corrupted or measuring some local event.

4.5 Experimental Evaluation

To evaluate the performance of ECLUN, we performed a set of experiments to test the effectiveness of each feature of ECLUN, and to compare the performance of ECLUN with the state-of-the-art competing algorithm, Snapshot Queries [Kot05]. We start by describing our real and synthetic datasets in Section 4.5.1, then our

evaluation methodology for each set of experiments in Section 4.5.2, in Section 4.5.3 we describe the settings of our experiments and then we conclude this section by discussing the experimental results in Section 4.5.4.

4.5.1 Datasets

We have used three real datasets in addition to one synthetic dataset for evaluating ECLUN. We give a description of each with some of the parameter settings applied with them on both ECLUN and Snapshot Queries. Unless otherwise stated, these parameter settings applies to all experiments.

Real Datasets

We used three real datasets:

Intel Berkeley Research Lab (Intel Lab 1) [Dat04]: Out of the 54 nodes readings collected in [Dat04]. Three nodes had a huge number of missing readings, therefore we used the clean readings of 51 nodes each contains 4-parameters readings taken every 31 seconds. The clean processed dataset contained 15730 readings. We have mapped these time stamps into 5 days, 15 hours, 27 min and 10 sec period of time. The network topology was selected to be as close as possible to original nodes topology. When applying this dataset on any algorithm we set the initial energy IE of each node to 295 Joules. We call this real dataset *Intel Lab 1* in our next experiments.

Intel Berkeley Research Lab (Intel Lab 2): To get more readings, we have excluded 5 nodes from the original Intel lab dataset. This resulted in 23077 healthy readings. For small missed values in between, we have always inserted the last received value instead of later missed readings. Again we set the topology of the network in both evaluated algorithms to be as close as possible to the original topology. When applying this dataset on any algorithm we set the initial energy IE of each node to 10000 Joules. We call this dataset *Intel Lab 2*.

I9 Sensor Dataset: A detailed explanation about this one-dimensional dataset with 40589 readings can be found in Section 2.5 and in [HMS09]. The 16 nodes were randomly inserted to the algorithms without mapping the coordinates of network topology. When applying this dataset on any algorithm we set the initial energy IE of each node to 1100 Joules.

Synthetic Dataset

The synthetic dataset was generated mainly for evaluating the response of each of the competing algorithms to some inserted changes in the monitored phenomena. We generated readings for 49 sensor nodes distributed in one 7x7 grid with 12000 3-dimensional readings for each node. The normally distributed random readings were mainly simulating the humidity, light and temperature attributes sensed by TelosB nodes [PSC05]. The total range of each attribute was divided into three subranges: Low, normal and High. Inserted events are any combination of three ranges, each taken from an attribute. We have generated 2 different events in different parts of the network, details about these events are depicted in Table 4.1.

	Event 1	**Event 2**
Values per dimension	D1{Low} D2{High} D3{Low}	D1{High} D2{Low} D3{High}
Time stamps [From, to]	[0,200], [1000,1100], [10000,11000]	[300,350], [1000,1100], [2000,2500]
Most affected node	Node 2, coordinates: (2,0)	Node 47, coordinates: (4,6)

Table 4.1: Generated events in the synthetic dataset

4.5.2 Evaluation Methodology

We evaluated ECLUN from three different perspectives:

1. **Evaluating each Feature of ECLUN:** We have tested the effect of each feature of ECLUN by evaluating for every feature two versions of ECLUN, one containing this feature and the other not. The measure was the total number of dead nodes in the whole network with the progress of time.

2. **Energy Consumption:** Two measures were performed to evaluate the energy consumption of ECLUN with that of Snapshot Queries, the total number of dead nodes in the network, and the total amount of consumed energy in Joules according to the energy cost model presented in Section 2.3.5.

3. **Detection of Changes in Input Stream:** We wanted to see the values of readings delivered to the server by the representatives in each algorithm on time stamps where we synthetically inserted events as in Table 4.1.

4.5.3 Setup of the Experiments

For evaluating the energy consumption, in all experiments we used the energy model described in [HMS09]. For all experiments of ECLUN, we had the following settings on all datasets: the radius of covered nodes by the range of each node: $\varepsilon = 2$, the maximum number of nodes in one physical cluster: $MaxNds = 4$ and the delegation authority threshold: 50% of initial energy. For **Intel Lab 1** and **Intel Lab 2** datasets, we have selected the number of initial readings $l = 10$, the threshold of relevant attributes for representing $Min\_Rel\_Attr = 2$, the node-representative update thresholds: δ_j; $j = (1 \cdots 4)$ as $(0.2, 0.2, 0.2, 0.2)$ and the representative-server update thresholds: ψ_j; $j = (1 \cdots 4)$ as $(0.2, 0.2, 0.2, 0.2)$. For **I9** dataset: $l = 10$, $Min\_Rel\_Attr = 1$, $\delta_1 = 0.2$ and $\psi_1 = 0.2$. To have fair results, the parameter settings of Snapshot Queries were always identical to that of ECLUN whenever they apply. We set the error threshold T_j; $j = (1 \cdots 4)$ to $(5, 5, 5, 5)$. According to [Kot05], these values deliver the best results in terms of number of participating nodes in each cluster on the one hand, and an accepted representation error on the other hand.

4.5.4 Experimental Results

Results of Features Evaluation

In each of the following selected two experiments, we test a feature in ECLUN, by comparing the energy consumption of two versions of ECLUN that differ only in including this feature or not.

Node - Representative and Representative - Server Communications: This feature enables the update of the representative or the server to occur only when a change is detected. Excluding it means that the nodes always communicate with the representative whenever they have a new reading, and the representative in turn always communicates with the server. As expected, the results in Figure 4.3 shows that this feature extends the whole network life time. Without using this feature, nodes start to die in the network after 5 days, 14 hours, 1

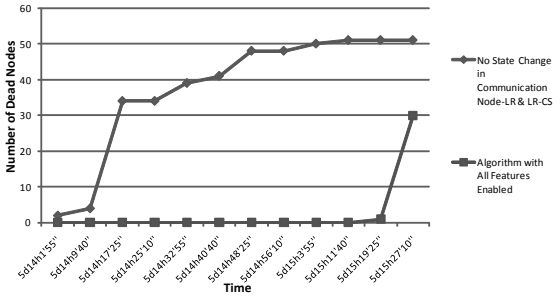


Figure 4.3: Testing the ECLUN feature of performing the update only when a change is detected using the Intel Lab 1 dataset

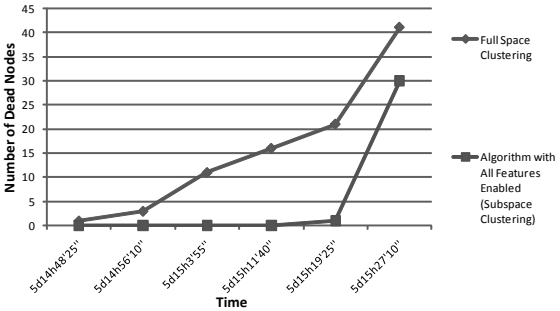


Figure 4.4: Testing the ECLUN feature of subspace clustering using the Intel Lab 1 dataset

minute and 55 seconds, while by using it, the first node dies around 1 hour and 20 minutes after that. Additionally, by the end of the dataset, 21 nodes are still alive when enabling this feature, while all nodes die when disabling it. As we will see in the change detection results, this feature is not delaying important changes.

Subspace Clustering (Clustering per Relevant Attributes): Disabling this feature means that a node can only be represented by nodes that are relevant to it in all spaces (attributes). The possibility for nodes to find such a representative in its neighbors will be very low. Which ends with a self representation by the node. As depicted in Figure 4.4, using this feature delays the death of first node around 31 minutes and increases the number of the still-alive nodes by 11 with the end of the simulation. The impact of the subspace clustering is even stronger

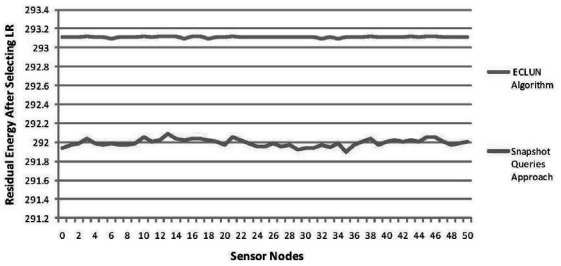


Figure 4.5: The residual energy in each of the 51 nodes after the process of selecting representatives in ECLUN and Snapshot Queries using Intel Lab 1

with higher dimensions.

Results of Energy Consumption Evaluation

We evaluate here the energy consumption of ECLUN and Snapshot Queries algorithm [Kot05]. Figure 4.5 depicts the residual energy in Joules of each sensor node after the initialization phase. As shown, the initialization phase of Snapshot Queries consumes more energy than that of ECLUN. This is because of the extensive messages of big sizes that are exchanged between nodes in Snapshot Queries during this phase. Although this initialization phase does not happen so often in ECLUN through reclustering, our experiments showed that it happens more likely in Snapshot Queries. This is due to the subspace nature that ECLUN uses. It can be seen also in Figure 4.5, that the energy consumption in ECLUN is balanced between all the nodes after this phase, in contrast to Snapshot Queries, where selected representatives consume more energy than others even during this phase.

Figure 4.6 presents a comparison between two variants of each algorithm. For ECLUN, we used the all-features variant and another without the previous features tested in 4.5.4 and without the delegation of authority optimization. For Snapshot Queries, we applied the two forms of changing the representative with the decrease of energy suggested in [Kot05]. The first one is similar to ECLUN, where nodes are invited to send their residual energy and the one with the highest residual energy is selected. The other approach randomly selects the next representative, we called this (Snapshot Queries with randomized representatives). The two variants of ECLUN extend considerably the network life time much more than both of the variants of Snapshot Queries. The better variant of

Snapshot Queries starts to lose nodes around 7 hours and 15 minutes earlier than the normal ECLUN. When the dataset ends, ECLUN has still 21 alive nodes, while the two variants of Snapshot Queries almost lose all of their nodes. Another important feature is that the nodes in ECLUN die close to each other, which yields a better usage of the network resources and more data about observed phenomena. Figure 4.7 and Table 4.2 show the efficiency of ECLUN over Snapshot Queries for different sizes of data with different dimensionality.

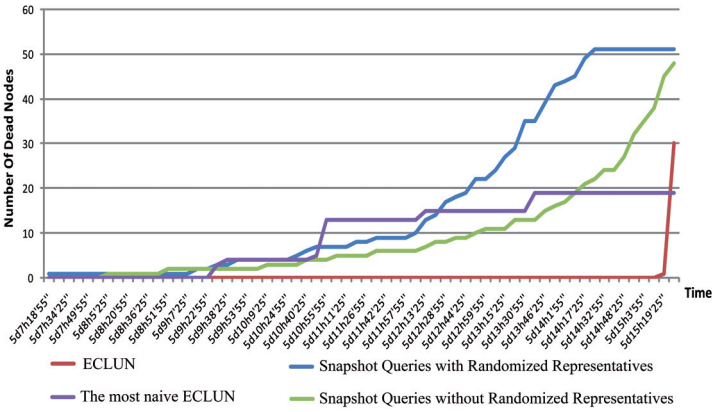


Figure 4.6: Number of dead nodes in different versions of ECLUN and Snapshot Queries using Intel Lab 1

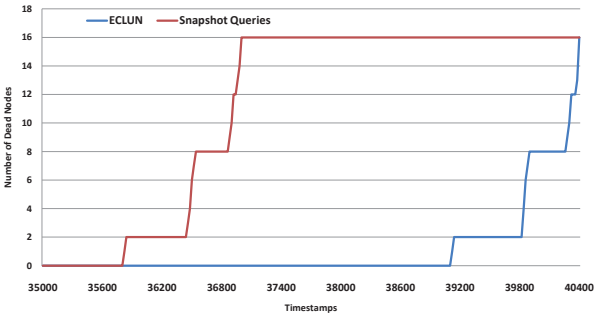


Figure 4.7: Number of dead nodes using the I9 dataset

Dataset	Number of Nodes	Number of Readings per Node	**ECLUN** Energy Consumption [Joules]	**Snapshot** Energy Consumption [Joules]
Intel Lab 2	49	23077	**22552.5**	25546.9
I9	16	40589	**13837.1**	14094.9

Table 4.2: Total energy consumption of ECLUN and Snapshot queries in Joules

Results of Change Detection Evaluation

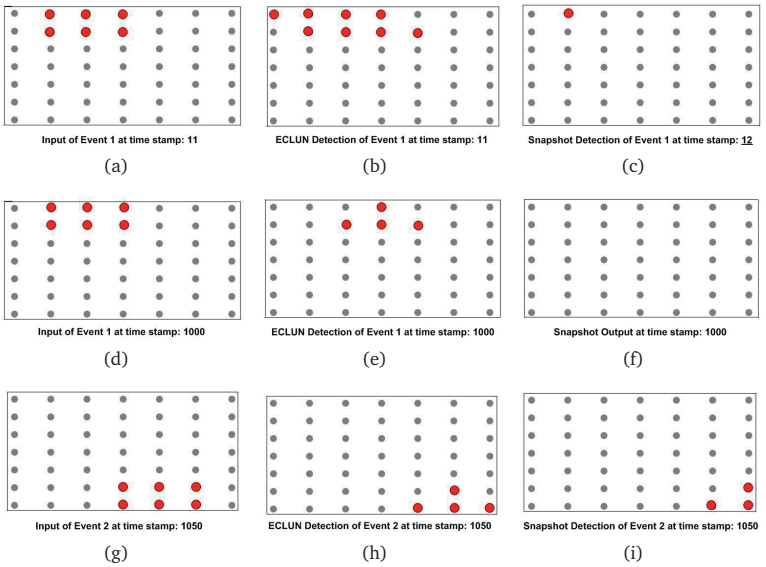


Figure 4.8: Change detection evaluation using the Synthetic Dataset with 2 inserted events

For evaluating this measure, we used the synthetic dataset. Figure 4.8 depicts the input events affecting some parts of the sensor network, and the corresponding output sent by ECLUN and Snapshot Queries to the server at the same time stamp. Figures 4.8(a) and 4.8(b) show the input and ECLUN output Event 1 at time stamp:11. Snapshot Queries detected this event at time stamp:12 Figure 4.8(c). Obviously, ECLUN was not only able to detect this event exactly when it appeared, it could also deliver the involved nodes in this event with few false

positives. Snapshot Queries detected the change event with a delay of one time stamp, then delivered the data of only one node out of the six involved in Event 1. Figures 4.8(d), 4.8(e) and 4.8(f) describe the input, ECLUN output and Snapshot Queries output at time stamp:1000. ECLUN detected the event at the same time stamp but was less accurate than at time stamp:11. This is due to the fact that at time stamp:11, ECLUN has clustered the nodes according to the first $l = 10$ readings. Event 1 was existing during that interval, and thus detecting it was much more accurate. Snapshot Queries could not detect this event at all. Figures 4.8(g), 4.8(h) and 4.8(i) depict the same sequence for Event 2 at time stamp: 1050.

4.6 Conclusion

In this chapter, we proposed a novel algorithm for an energy-aware physical clustering of sensor nodes. Our algorithm considers both spatial and data similarities when building these physical clusters. Nodes in our suggested approach make use of established data mining techniques like subspace clustering for joining physical clusters according to relevant attributes, and outlier detection for online exclusion of outlying readings. We further suggested a powerful method for the maintenance of the constructed clusters. This enables the network to adapt with different changes of observed phenomena in an unsupervised way, while consuming less energy. We proved the efficiency and effectiveness of our approach through comprehensive experiments.

Part II

High-Dimensional Density-Based Stream Clustering

Chapter 5

Projected Density-based Stream Clustering

* In this chapter, we present, deeply analyze, and experimentally validate our novel subspace data stream clustering algorithm, termed *PreDeConStream* (subspace **Pre**ference weighted **De**nsity **Con**nected clustering of **Stream**ing data). The technique is based on the two-phase model of clustering streaming data, where the first phase represents the process of the online maintenance of a data structure, that is then passed to an offline phase for generating the final clustering. The technique works on incrementally updating the output of the online phase stored in a microcluster structure. Taking those microclusters that are fading out over time into consideration speeds up the process of assigning new data points to existing clusters. A density-based projected clustering model in developing *PreDeConStream* was used. Localizing the change of the previous clustering result, in addition to the smart usage of the clustering validity interval made our model efficient in its offline phase. With many important applications that can benefit from such technique, we have proved experimentally the superiority of the proposed methods over several state-of-the-art techniques.

5.1 Motivation

Data streams represent one of the most famous forms of the massively complex data. The continuous and endless flow of streaming data results in a huge amount of data on the one hand, and the constant change of the data distribution, on the

*Parts of this chapter have been published in the Proceedings of the 6th International Conference on Scalable Uncertainty Management (SUM 2012) [HSGS12].

other hand, cause a high complexity of the data. The most famous examples of complex data streams are sensor streaming data which are available in everyday applications. These applications start from home scenarios like the smart homes to environmental applications and monitoring tasks in the health sector [HS11, QBG+13], but do not end with military and aerospace applications.

In many applications of streaming data, objects are described by using multiple dimensions (e.g. the Network Intrusion Dataset [Dat99] has 42 dimensions). For such kinds of data with higher dimensions, distances grow more and more alike due to an effect termed *curse of dimensionality* [BGRS99] (cf. the toy example in Figure 5.1). Applying traditional clustering algorithms (called in this context: *full-space* clustering algorithms) over such data objects will lead to useless clustering results. In Figure 5.1, the majority of the black objects will be grouped in single-object clusters (outliers) when using a full-space clustering algorithm, since they are all dissimilar, but apparently they are not as dissimilar as the gray objects. If we consider $Dim2$ only, we will notice that unlike the gray objects, the black ones will form a dense cluster (Cluster 2). Thus, considering subsets of the dimensions, might result in discovering a similarity between the objects that is hidden when considering the full space. This is much more likely to happen with high-dimensional data than in the $2d$ toy example in Figure 5.1. The latter fact motivated the research in the domain of *subspace* and *projected clustering* in the last decade which resulted in an established research area for static data.

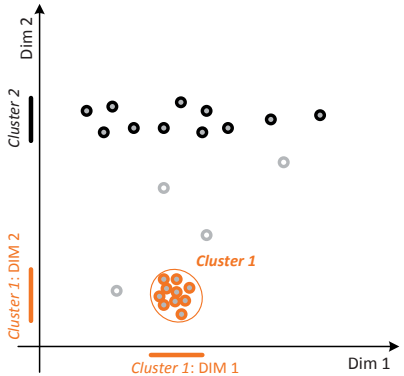
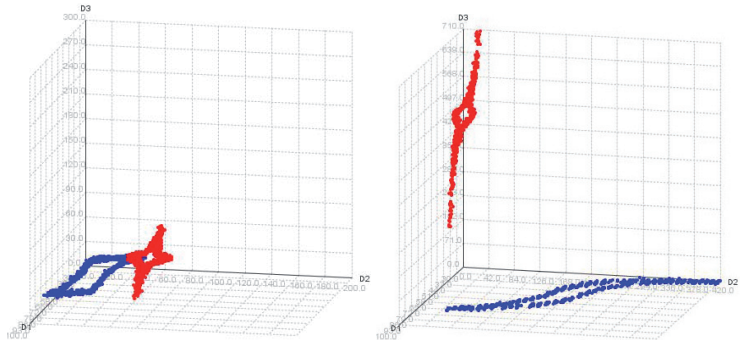


Figure 5.1: An example of subspace clustering for a static $2d$ dataset.

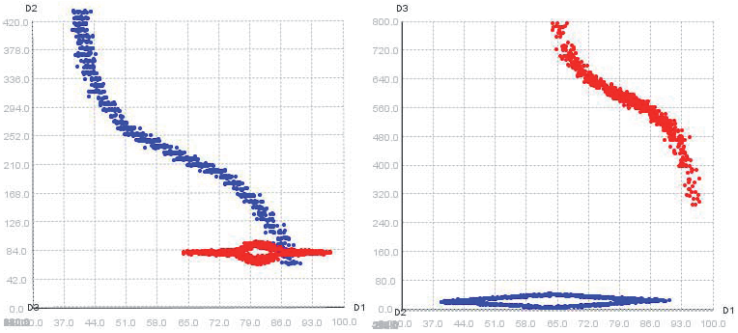


(a) (b)

Figure 5.2: An example of a change in the *SynStream3D* streaming dataset (cf. Section 5.5.1): (a) before the change: the two clusters are detected, (b) after the stream evolves, changes: Dimension $D3$ becomes irrelevant for the red cluster and Dimension $D2$ becomes irrelevant of the blue cluster. A full-space stream clustering algorithm will assign all points after the change as outliers, while, in fact, the red points are forming a cluster when considering the Subspace: $(D1, D2)$ (cf. Figure 5.3(a)), and the blue points are forming a cluster when considering the Subspace: $(D1, D3)$ (cf. Figure 5.3(b)).

Figure 5.2 depicts another example but for a $3d$-data stream, before and after the stream evolves such that some dimensions become irrelevant for both "clusters". Figure 5.3 shows that projections of these two "clusters" can still form some clusters after the change. A full-space stream clustering algorithm will consider all the data points after the change as outliers, because it considers all dimensions when calculating the distances between objects. In streaming data, although a considerable research has tackled the full-space clustering (cf. Section 5.2.2), very limited work has dealt with subspace clustering (cf. Section 5.2.3).

Most full-space stream clustering algorithms use a two-phased (online-offline) model (e.g CluStream [AHWY03] and DenStream [CEQZ06], cf. Section 5.2.2). While the online part summarizes the stream into groups called *microclusters*, the offline part performs some well-known clustering algorithm over these summaries and gives the final output as clustering of the data stream. Usually, the offline part represents the bottleneck of the clustering process, and when considering the projected clustering, which is inherently more complicated compared to the full-space clustering, the efficiency of a projected or subspace stream clustering algorithm becomes a critical issue. In this chapter we present a density-based



(a) (b)

Figure 5.3: Two projections of the *SynStream3D* dataset after the change (cf. Figure 5.2(b)) above: (a) shows that the red points are forming a cluster in the Subspace: $(D1, D2)$ while the blue points are noise (mind the scaling on $D2$), (b) shows that the blue points are forming a cluster in the Subspace: $(D1, D3)$ while the red points are noise (mind the scaling on $D3$).

projected clustering over streaming data. Our algorithm *PreDeConStream*, tries to find clusters over subspaces of the evolving data stream instead of searching over the full space merely. The algorithm uses the famous (online-offline) model, where in the offline phase, it efficiently maintains the final clustering by localizing the part of the clustering result which was affected by the change of the stream input within a certain time, and then sustaining only that part. Additionally, the algorithm specifies the time intervals within which a guaranteed no-change of the clustering result can be given.

The remainder of this chapter is organized as follows: Section 5.2 gives a short overview of the related work from different neighboring areas. Section 5.3 introduces some required definitions and formulations to the problem. Our algorithm PreDeConStream is introduced in Section 5.4 and then thoroughly evaluated in Section 5.5. Then we conclude the chapter in Section 5.6.

5.2 Related Work

In this section, we list the related work from three areas: subspace clustering of static data, full-space stream clustering, and finally subspace stream clustering.

5.2.1 Subspace Clustering Algorithms over Static Data

According to [KKZ09], one can differentiate between two main classes of subspace clustering algorithms that deal with static data:

- ***Subspace* clustering algorithms** [KKK04, AWY+99, LXY00], that aim at detecting all possible clusters in all subspaces. In this algorithm class, each data object can be part of multiple subspace clusters.

- ***Projected* clustering algorithms** [BKKK04, AGGR98, CFZ99, NGC01], that assign each data object to at most one cluster. For each cluster, a subset of projected dimensions is determined which represents the projected subspace.

SubClu [KKK04] is a subspace clustering algorithm that uses the DBSCAN [EKSX96] clustering model of density connected sets. SubClu computes for each subspace all clusters which DBSCAN would have found as well if applied on that specific subspace. The subspace clusters are generated in a bottom-up way and a monotonicity criteria [KKK04] is used for the sake of efficiency. If a subspace T does not contain a cluster, then no higher subspace S with $T \subseteq S$ can contain a cluster.

PreDeCon [BKKK04] is a projected clustering algorithm which adapts the concept of density-based clustering [EKSX96]. It uses a specialized similarity measure based on the subspace preference vector (cf. Equation 5.10) to detect the subspace of each cluster. Different to DBSCAN, a preference weighted core point is defined in PreDeCon as the point whose number of preference dimensions is at most λ and the preference weighted neighborhood contains at least μ points.

IncPreDeCon [KKNZ10] is an incremental version of the algorithm PreDeCon [BKKK04] designed to handle accumulating data. It is unable to handle evolving stream data since it does not perform any removal or forgetting of aging data. Additionally, the solution performs the maintenance after each insertion, which makes it considerably inefficient, especially for applications with limited memory. The algorithm we present in this chapter adopts in some parts of its offline phase the insertion method of IncPreDeCon, but fundamentally differs from IncPreDeCon by maintaining the summaries of drifting streaming data, applying PreDeCon on the microcluster level, including a novel deletion method, and carefully performing the maintenance of the clustering after some time interval and not after each receiving of an object.

5.2.2 Full-space Clustering Algorithms over Streaming Data

There is a rich body of literature on stream clustering. Approaches can be categorized from different perspectives, e.g. whether convex or arbitrary shapes are found, whether data is processed in chunks or one at a time, or whether it is a single algorithm or it uses an online component to maintain data summaries and an offline component for the final clustering. Convex stream clustering approaches are based on a k-center clustering [AHWY03]. Detecting clusters of arbitrary shapes in streaming data has been proposed using kernels [JZC06], fractal dimensions [LC08] and density-based clustering [CEQZ06, CT07, Agg07, HTGS14]. Another line of research considers the anytime clustering with the existence of outliers [HKS11]. Since DenStream [CEQZ06] is the most appealing density-based *full-space* stream clustering algorithm, we compared our algorithm against it (cf. Section 5.5.3).

5.2.3 Subspace Clustering Algorithms over Streaming Data

Similar to the offline clustering algorithms, two types of stream clustering algorithms exist: subspace and projected stream clustering algorithms. However, there is, to the best of our knowledge, only one subspace clustering algorithm and two projected clustering ones over streaming data.

Sibling Tree [PL07] is a grid-based *subspace* clustering algorithm where the streaming distribution statistics are monitored by a list of grid-cells. Once a grid-cell is dense, the tree grows in that cell in order to trace any possible higher dimensional cluster.

HPStream [AHWY04] is a *k*-means-based *projected* clustering algorithm for high dimensional data stream. The relevant dimensions are represented by a d-dimensional bit-vector D, where 1 marks a relevant dimension and 0 otherwise. HPStream uses a projected distance function, called Manhattan Segmental distance MSD [AWY+99], to determine the nearest cluster. HPStream cannot detect arbitrary cluster shapes and a parameter for the number of cluster k has to be given by the user, which is not intuitive in most scenarios. Additionally, as a *k*-means-based approach, HPStream is a bit sensitive to outliers. The model described in this chapter is able to detect arbitrarily shaped and numbered clusters in subspaces and, due to its density-based method, is less sensitive to outliers. Being a standard for the k-means-based projected stream clustering algorithms, we compared our algorithms against HPStream w.r.t. both clustering quality (cf

Section 5.5.3) and efficiency (cf. Section 5.5.3).

HDDStream [NZP+12] is a recent density-based projected stream clustering algorithm that was developed simultaneously with PreDeConStream. HDDStream performs an online summarization of both points and dimensions and then, similar to PreDeConStream it performs a modified version of PreDeCon in the offline phase. Different from our algorithm, HDDStream does not optimize the offline part which is usually the bottleneck of subspace-stream clustering algorithm. In the offline phase, our algorithm localizes effects of the stream changes and maintains the old clustering results by keeping non-affected parts. Additionally, our algorithm defines the time intervals where a guaranteed no-change of the clustering result exists, and organizes the online summaries in multiple lists for a faster update. This makes our algorithm considerably faster than HDDStream as we extensively show in the experimental part (cf. Section 5.5.3), while delivering projected clusters of the same or even better quality (cf. Section 5.5.3).

5.3 Problem Formulation and Definitions

In this section, we formulate our related problems and give some definitions and data structures that are needed to introduce our PreDeConStream algorithm. In its online phase, our algorithm adopts the microcluster structure [AHWY03], [CEQZ06] with an adaptation to fit our problem (cf. Definitions 5.2-5.5). Later, we introduce the data structure used for managing the online microclusters in Section 5.3.2. Finally, we introduce some definitions which are related to the offline phase (cf. Section 5.3.3).

Since our algorithm uses a density-based clustering over its online and offline phases, similar notations that appear in both phases are differentiated with an F subscript for the offline phase and N for the online phase.

5.3.1 Basic Definitions in the Online Phase

Definition 5.1 ***The Decaying Function*** *The fading function [CEQZ06] used in PreDeConStream is defined as $f(t) = 2^{-\lambda t}$, where $0 < \lambda < 1$ The weight of the data stream points decreases exponentially over time, i.e. the older a point gets, the less important it gets. The parameter λ is used to control the importance of the historical data of the stream.*

Definition 5.2 ***Core and Potential Microclusters*** *A core microcluster at time t is defined as a group of close, streaming, d-dimensional points $p_1, \ldots, p_n$ with timestamps $t_1, \ldots, t_n$. It is represented by a tuple $cmc(w, \overline{CF^1}, \overline{CF^2}, \bar{c}, r)$ with:*

1. *Weight, $w = \sum_{j=1}^{n} f(t - t_j)$, with $w \geq \mu_N$*
2. *d-dim. weighted linear sum of the points, $\overline{CF^1} = \sum_{j=1}^{n} f(t - T_j)\overline{p_j}$*
3. *d-dim. weighted squared sum of the points, $\overline{CF^2} = \sum_{j=1}^{n} f(t - T_j)\overline{p_j^2}$*
4. *d-dim. center, $\bar{c} = \frac{\sum_{j=0}^{n} f(t-t_j)\overline{p_j}}{w} = \frac{\overline{CF^1}}{w}$*
5. *Radius, $r = \sqrt{\frac{|\overline{CF^2}|}{w} - \left(\frac{|\overline{CF^1}|}{w}\right)^2}$, with $r \leq \varepsilon_N$*

Where the decision of how close the points are, is done using the Euclidean distance function and a maximal radius threshold of cmc.

Where μ_N represents the minimum weighted number of points needed to be within the ε_N-neighborhood of c in order to make the current microcluster a core microcluster.

A **potential** microcluster is defined similarly with a tuple $pmc = (\overline{CF^1}, \overline{CF^2}, w, \bar{c}, r)$ with only a difference that it is enough that $w \geq \beta\mu_N$ with $0 < \beta \leq 1$. The weight and the statistical information about the stream data decay according to the fading function (cf. Definition 5.1). The maintenance of the microclusters is discussed in Definition 5.4. An additional type of microclusters is also given, the outlier microcluster, to allow the algorithm to quickly recognize changes in the data stream.

Definition 5.3 ***Outlier microcluster*** *An outlier microcluster $omc = (\overline{CF^1}, \overline{CF^2}, w, \bar{c}, r, t_0)$ is defined as cmc and pmc with the following modifications:*

1. *Weight $w = \sum_{j=1}^{n} f(t - T_j)$ with $w < \beta\mu_N$, $0 < \beta \leq 1$*
2. *An additional entry about the creation time t_0, to decide whether the outlier microcluster is being evolving or is fading out.*

The parameter β controls how tolerant the algorithm is to including outdated microclusters.

Definition 5.4 ***Microclusters Maintenance*** *With the evolution of the stream, any core, potential or outlier microcluster at time t:*
$mc_t = (\overline{CF^1}, \overline{CF^2}, w)$ is maintained as follows: If a point p hits mc at time $t+1$ then its statistics become: $mc_{t+1} = (2^{-\lambda} \cdot \overline{CF^1} + p, 2^{-\lambda} \cdot \overline{CF^2} + p^2, 2^{-\lambda} \cdot w + 1)$ Otherwise, if no point was added to mc for any time interval δt, the microcluster can be updated after any time interval δt as follows: $mc_{t+\delta t} = (2^{-\lambda \delta t} \cdot \overline{CF^1}, 2^{-\lambda \delta t} \cdot \overline{CF^2}, 2^{-\lambda \delta t} \cdot w)$.

It should be noted that this updating method is different from that in DenStream [CEQZ06]. The modification considers the decaying of the other old points available in mc, even if mc was updated. This makes the algorithm faster in adapting to the evolving stream data. Additionally, this gives our microcluster structure an upper bound for the weight (W_{max}) of the microcluster which will be useful for the maintenance of the offline part as we will see in Section 5.3.2.

Lemma 1 *The maximum weight W_{max} of any microcluster mc is $\frac{1}{1-2^{-\lambda}}$.*

Proof 5.1 *Assuming that all the points of the stream hit the same microcluster mc. The definition of the weight $w = \sum_{t'=0}^{t} 2^{-\lambda(t-t')}$ can be transformed with the sum formula for geometric series as following:*

$$w = \sum_{t'=0}^{t} 2^{-\lambda(t-t')} = \frac{1 - 2^{-\lambda(t+1)}}{1 - 2^{-\lambda}} \tag{5.1}$$

Thus, the maximum weight of a microcluster is:

$$W_{max} = \lim_{t \to \infty} w = \lim_{t \to \infty} \frac{1 - 2^{-\lambda(t+1)}}{1 - 2^{-\lambda}} \tag{5.2}$$

$$W_{max} = \frac{1}{1 - 2^{-\lambda}} \tag{5.3}$$

Any newly created microcluster needs a minimum time T_p to grow into a potential microcluster, during this time the microcluster is considered as an outlier microcluster. Similarly, there is a minimum time T_d needed for a potential microcluster to fade into an outlier microcluster.

Lemma 2 ***A)*** *The minimum timespan for a newly created microcluster to grow into a potential microcluster is: $T_p = \left\lceil \frac{1}{\lambda} \log_2 \left(\frac{1}{1-\beta\mu_N(1-2^{-\lambda})} \right) - 1 \right\rceil$.*
B) *the minimum timespan needed for a potential microcluster to fade into an outlier microcluster is: $T_d = \lceil \frac{1}{\lambda} \log_2(\beta\mu_N) \rceil$.*

Proof 5.2 ***A)*** *The minimum timespan needed for a newly created microcluster to become potential is:*

$$T_p = t_p - t_0$$

where t_p is the first timestamp where the microcluster becomes potential and t_0 the creation time of the outlier microcluster.

According to Definition 5.2, a microcluster becomes potential when its weight w becomes $w \geq \beta\mu_N$. Thus, from Equation 5.1:

$$w = \sum_{t'=t_0}^{T_p} 2^{-\lambda(t-t')} \tag{5.4}$$

$$w = \frac{1 - 2^{-\lambda(T_p+1)}}{1 - 2^{-\lambda}} \geq \beta\mu_N \tag{5.5}$$

$$\Rightarrow T_p = \left\lceil \frac{1}{\lambda} \log_2 \left(\frac{1}{1 - \beta\mu_N(1 - 2^{-\lambda})} \right) - 1 \right\rceil . \tag{5.6}$$

B) *Let the minimum timespan needed for a potential microcluster to be deleted, be:*

$$T_d = t_d - t_p$$

where t_p is the last timestamp where the microcluster was still potential, and t_d is the time when it is deleted. For the deletion, the weight of an outlier microcluster has to be less than $W_{min} = 1$, because the start weight of a newly created microcluster is 1. Let w_p be the last time when the microcluster was potential.

According to Definition 5.4, T_d is the smallest no-hit interval that is needed for a potential microcluster to become outlier. Thus: T_d is the smallest value which makes:

$$w_p \cdot 2^{-\lambda T_d} < 1$$

But we know from Definition 5.2 that:

$$w_p = \beta\mu_N$$

$$\Rightarrow T_d = \left\lceil \frac{1}{\lambda} \log_2(\beta\mu_N) \right\rceil . \tag{5.7}$$

Definition 5.5 ***Minimum Offline Clustering Validity Interval*** *The minimum validity interval of an offline clustering T_v defines the time within which PreDeCon-Stream does not need to update the offline clustering since it is still valid because no change of the status of any microcluster status happened. It is defined as:*

$T_v = \min\{T_p, T_d\}$

For a correct non-negative value of T_v, the parameter β should be correctly selected. Since $\beta\mu_N$ defines the weight threshold that differentiates between the potential and core microclusters, it does not make sense to select $\beta\mu_N$ to be greater than the maximal weight of a microcluster (cf. Lemma 1). Additionally, it makes also no sense to select $\beta\mu_N$ to be less than $w_d = 1$, because it is the threshold for deleting an outlier microcluster. Therefore the lower and upper bounds for β are defined as follows:

$$\frac{1}{\mu_N} < \beta < \frac{1}{(1 - 2^{-\lambda})\mu_N} \tag{5.8}$$

5.3.2 A Data Structure to Manage the Microclusters

A data structure is needed to manage the updated and non-updated microclusters at each timestamp in an efficient and effective way. The main idea is that the algorithm does not need to check for all *potential* microclusters, at each timestamp, whether the potential microcluster remains potential or fades into a *deleted* microcluster. Therefore a data structure is introduced where only a subset of all the potential microclusters needs to be checked.

We group the microclusters into multiple lists according to their weight. The borders between these lists are selected as below (cf. also Figure 5.4) such that all microclusters in a list that are not hit in the previous timestamp will fade to the lower-weighted list. Thus, only the weight of the one which was hit needs to be checked. There are two types of lists: outlier lists l_j^o, and potential lists: l_i^p. Core microclusters are special cases of potential microclusters, and only both of them are considered in the offline phase. The borders of the lists are:

$$W_d = 1, W_{min} = \beta\mu_N, W_{max} = \frac{1}{1 - 2^{-\lambda}}$$

The internal borders for the potential lists are selected as:

$$w_i^p = \frac{w_{i-1}^p}{2^{-\lambda}}$$

and the internal borders of the outlier lists are selected as:

$$w_i^o = 2^{-\lambda} w_{i-1}^o + 1$$

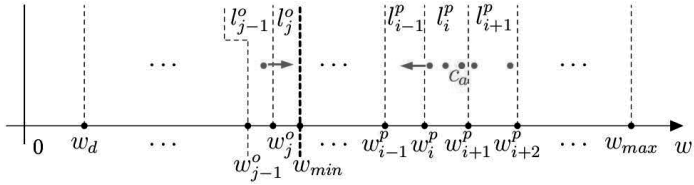


Figure 5.4: An example of outlier and potential lists visualized w.r.t. their weights.

It should be noted, that in this case, only the lists around W_{min} from both outlier and potential sides (cf. Figure 5.4) need to be checked each T_v to see whether the current offline clustering is still valid as we will see in Section 5.4. Only populated lists do exist in the memory. Once a list is not any more populated, it is deleted, and it is recreated once it is populated again. Since a list can contain many microclusters, the number of lists is manageable and considerably smaller than the number of microclusters.

5.3.3 Basic Definitions in the Offline Phase

Now that we have given the definitions for the creation, maintenance and indexing online microclusters in Section 5.3.1, we will introduce in this section the definitions needed for finding the final offline projected clusters out of these microclusters. These definitions are mainly inspired by the ones introduced in [BKKK04].

Let *CMC* and *PMC* be at time t the set of core and potential microclusters, respectively. For each core microcluster $c_p \in$ *CMC*, a *preference subspace vector* V_{c_p} is determined. All the potential and core microclusters c_q which belong to the ε_F-neighborhood of c_p are determined and for each dimension i of that neighborhood, the variance $VAR_i(\mathcal{N}_{\varepsilon_F}(c_p))$ is computed as follows:

$$VAR_i(\mathcal{N}_{\varepsilon_F}(c_p)) = \frac{\sum_{c_q \in \mathcal{N}_{\varepsilon_F}(c_p)} dist(c_p[i], c_q[i])^2}{|\mathcal{N}_{\varepsilon_F}(c_p)|} \tag{5.9}$$

Where $dist(c_p[i], c_q[i])$ represents the *normal* Euclidean distance between $c_p[i]$ and $c_q[i]$. If the variance $VAR_i(\mathcal{N}_{\varepsilon_F}(c_p))$ of the dimension i is below a user defined threshold δ, then the i-th entry of the preference subspace vector V_{c_p} is set to a constant $\kappa \gg 1$ to identify i as a "relevant" dimension. Otherwise, (i.e. if the

microclusters within the ε_F-neighborhood of c_p were sparse over the dimension i), then the entry $V_{c_p}(i)$ is set to 1. An example is shown in Figure 5.5.

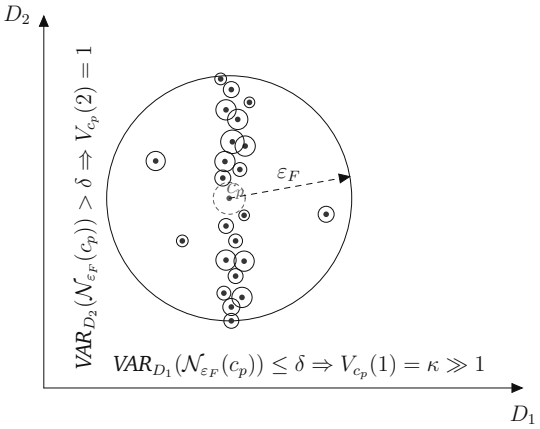


Figure 5.5: An example on the computation of the subspace preference vector: $V_{c_p} = (\kappa, 1)$

A weighted Euclidean distance function is used to determine the preference weighted ε_F-neighborhood $\mathcal{N}_{\varepsilon_F}^{V_{c_p}}(c_p)$.

$$dist_{c_p}(c_p, c_q) = \sqrt{\sum_{i=1}^{d} V_{c_p}(i) \cdot (c_p[i] - c_q[i])^2} \tag{5.10}$$

where $V_{c_p}(i)$ is the i-th entry of the preference subspace vector V_{c_p}.

The similarity measure in Equation 5.10 is not symmetric, due to the possible differences between V_{c_p} and V_{c_q}, therefore, the maximum of both values is chosen:

$$dist_{pref}(c_p, c_q) = \max\{dist_{c_p}(c_p, c_q), dist_{c_q}(c_q, c_p)\} \tag{5.11}$$

Based on the above preference weighted distance measure, the preference *weighted* ε_F-neighborhood $\mathcal{N}_{\varepsilon_F}^{V_{c_p}}(c_p)$ can now be defined as follows:

$$\mathcal{N}_{\varepsilon_F}^{V_{c_p}}(c_p) = \{c_q \in CMC \cup PMC \mid dist_{pref}(c_p, c_q) \leq \varepsilon_F\} \tag{5.12}$$

In Figure 5.6, the preference weighted ε_F-neighborhood using the preference

weighted distance measure in Equation 5.11 is depicted.

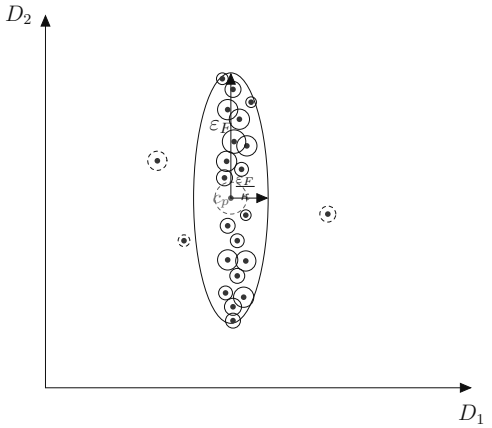


Figure 5.6: Weighted Euclidean ε_F-neighborhood of microcluster c_p: $\mathcal{N}_{\varepsilon_F}^{V_{c_p}}(c_p)$

Definition 5.6 (Preference Weighted Core P-Microcluster) *A preference weighted* ***core*** *p-microcluster is a potential microcluster $c_p \in CMC \cup PMC$ w.r.t $\varepsilon_F, \mu_F, \delta$ and τ, denoted by $CoreMC(c_p)$, which satisfies the following conditions:*

1. *The preference dimensionality of its $epsF$-neighborhood (denoted by$P(V_{c_p})$) is at most τ (i.e. the number of entries that contain "1" in V_{c_p} is maximally τ)*

2. *Its weighted ε_F-neighborhood contains at least μ_F p-microclusters:*

$$CoreMC(c_p) \Leftrightarrow P(V_{c_p}) \leq \tau \wedge |\mathcal{N}_{\varepsilon_F}^{V_{c_p}}(c_p)| \geq \mu_F \tag{5.13}$$

It should be noted that a $CoreMC$ should not be mixed up with the *online* core microcluster cmc introduced in Definition 5.2.

Definition 5.7 (Direct Preference Weighted Reachability) *The potential microclusters c_p and c_q are directly preference weighted reachable, denoted by $DirReach(c_p, c_q)$, if c_p is a preference weighted core microcluster and c_q is in the preference weighted ε_F-neighborhood of c_p. Furthermore the number of preference*

dimensions $P(V(c_q))$ *should be at most* τ.

$$\begin{aligned} DirReach(c_p, c_q) \Leftrightarrow & CoreMC(c_p) \\ & \wedge P(V(c_q)) \leq \tau \\ & \wedge c_q \in \mathcal{N}_{\varepsilon_F}^{V_{c_p}}(c_p) \end{aligned} \tag{5.14}$$

Please note that although c_q should satisfy also the condition: $P(V(c_q)) \leq \tau$, it is not required that $V(c_q) = V(c_p)$ in order for c_q to be directly reachable from c_p. Thus, in the offline phase, clusters with different relevant dimensions might be merged together if they contain direct preference weighted reachable microclusters.

Definition 5.8 (Preference Weighted Core Reachability) *The potential microclusters* c_p *and* c_q *are preference weighted reachable, denoted by* $Reach(c_p, c_q)$ *if the following condition holds*

$$Reach(c_p, c_q) \Leftrightarrow \exists c_{p_0}, \ldots, c_{p_n} \in \textit{CMC} \cup \textit{PMC} : DirReach(c_{p_i}, c_{p_{i+1}}) \tag{5.15}$$

with $0 \leq i < n$ *and* $c_p = c_{p_0}$ *and* $c_q = c_{p_n}$

Definition 5.9 (Preference Weighted Connectivity) *The potential microclusters* c_p *and* c_q *are preference weighted connected , denoted by* $Connect(c_p, c_q)$*, if there is a potential microcluster* c *such that both* c_p *and* c_q *are preference weighted reachable from* c

$$Connect(c_p, c_q) \Leftrightarrow \exists c \in \textit{CMC} \cup \textit{PMC with } Reach(c_p, c) \wedge Reach(c_q, c) \tag{5.16}$$

Figure 5.7 shows an example of preference weighted connected microclusters where the potential microclusters c_p and c_q are preference weighted connected by the potential microcluster c_o.

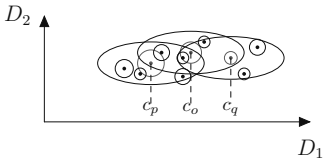


Figure 5.7: Preference weighted connected potential microclusters

Definition 5.10 (Subspace Preference Cluster) *A non-empty set: $C \subseteq$ CMC $\cup$ PMC is called a subspace preference cluster, w.r.t. $\varepsilon_F, \mu_F, \delta$ and τ if all the potential microclusters in C are preference weighted connected and C is maximal*

$$ConSet(C) \Leftrightarrow \textit{Connectivity: } \forall c_p, c_q \in C : Connect(c_p, c_q) \bigwedge$$
$$\textit{Maximality: } \forall c_p, c_q \in \textit{CMC} \cup \textit{PMC} : c_p \in C \wedge \quad (5.17)$$
$$Reach(c_p, c_q) \Rightarrow c_p \in C$$

Now that we have introduced all of our definitions from the online and the offline phases as well as the data structure where we index our online microclusters, we will present in the Section 5.4 our PreDeConStream algorithm.

5.4 The PreDeConStream Algorithm

Our algorithm is abstractly explained in Algorithm 5.1. It consists mainly of three phases: an initialization phase, an online phase, and an offline phase. In the following three subsections, we will explain each of those in details.

5.4.1 The Initialization Phase

In lines 1-4 of Algorithm 5.1, the minimum timespan T_v based on the user's parameter setting is computed. Furthermore, PreDeConStream needs an initial set of data stream points to generate an initial set of microclusters for the online part. Therefore a certain amount of stream data is buffered and on this initial data, all the points p whose neighborhood contains at least $\beta\mu_N$ weighted points in their ε_N-neighborhood, are detected. Whenever such a point p is found, a potential microcluster is created by p and all the points in its neighborhood, and they are removed from the initial points. This is repeated until no new potential microcluster is found. Finally the generated initial potential microclusters are inserted into the corresponding lists and the initial clustering is computed with an adapted version of PreDeCon [BKKK04].

Algorithm 5.1: PreDeConStream$(\mathrm{DS}, \varepsilon_N, \mu_N, \lambda, \varepsilon_F, \mu_F, \beta, \tau)$

1: $T_p \leftarrow \left\lceil \frac{1}{\lambda} \log_2 \left(\frac{1}{1-\beta\mu_N(1-2^{-\lambda})} \right) - 1 \right\rceil$;
2: $T_d \leftarrow \lceil \frac{1}{\lambda} \log_2(\beta\mu_N) \rceil$;
3: $T_v \leftarrow \min\{T_p, T_d\}$;
4: initialization phase
5: **repeat**
6: get next point $p \in \mathrm{DS}$ with the current timestamp t_c
7: *process(p)*; *// cf. Algorithm 5.2*
8: maintain microclusters in data structure; update the $Inserted\_PMC$ and $Deleted\_PMC$ accordingly
9: **if** $(t_c \mod T_v) == 0$ **then**
10: $C \leftarrow$ *updateClustering(C)*; *// cf. Algorithm 5.3*
11: **end if**
12: **if** user request clustering is received **then**
13: return clustering C
14: **end if**
15: **until** data stream terminates

5.4.2 Online: Maintenance of the Resulting Subspace Clustering

Line 6 of Algorithm 5.1 starts the online phase by picking a point from the stream, and it is then processed in Line 7. In Line 8 all the fading and the organization of microclusters into the correct lists is performed. Algorithm 5.2 details this *process(p)* step. First, the new arriving data points $p \in \mathrm{DS}$ of the stream data within timestamp t are merged with the existing microclusters. In Lines 1-4 of Algorithm 5.2, the nearest potential microcluster c_p is searched for in all the lists of the potential microclusters l^p. The algorithm clusters the incoming point $p \in \mathrm{DS}$ tentatively to c_p to check, if the point actually fits into the potential microcluster c_p. If the radius r_p of the temporary microcluster c_p is still less than ε_F, the point can be clustered into c_p without hesitation. If p does not fit into the nearest potential microcluster c_p, (cf. Lines 5-12 of Algorithm 5.2), the algorithm searches for the nearest outlier microcluster c_o in the outlier lists l^o. The algorithm checks again if its radius r_o of c_o is still less than ε_N, when the point is tentatively added to c_o. If the point fits into c_o and it is in the highest list of the outliers l^o, the algorithm checks if the weight w_o is greater than or equal to $\beta\mu_N$. If that is the case, the microcluster c_o is inserted into the lowest list $l^p_{\min}$ of the potential microclusters and has to be considered in the offline part. If the point does not fit into

Algorithm 5.2: process(data point p)

```
 1: search nearest potential microcluster c_p in all the lists l^p
 2: merge p tentatively into c_p
 3: if r_p ≤ ε_N then
 4:    insert p into c_p
 5: else
 6:    search nearest outlier microcluster c_o in all the lists l^o
 7:    merge p tentatively into c_o
 8:    if r_o ≤ ε_N then
 9:       insert p into c_o
10:       if w_o ≥ βμ_N then
11:          insert c_o into potential list l^p_min and remove it from outlier list l^o_max
12:       end if
13:    else
14:       create new outlier microcluster with p
15:    end if
16: end if
```

any existing microcluster, a new outlier microcluster is created with this point and is inserted into the outlier microcluster list l_0^o, (cf. Line 14).

5.4.3 Offline: Processing of the Data Stream

Lines 9-11 of Algorithm 5.1 contain the offline part of the PreDeConStream algorithm. Where the Clustering C is updated each minimum offline clustering validity interval T_v. The *updateClustering(C)* step is detailed in Algorithm 5.3. In Lines 1-10 of Algorithm 5.3, for each newly created potential microcluster c_p, its subspace preference vector w_{c_p} is computed. Furthermore, for each potential microcluster $c_q \in \mathcal{N}_{\varepsilon_F}(c_p)$, the preference subspace vector of each c_q is updated and checked if its core member property has changed. If that is the case, it is added to the $AFFECTED\_CORES_i$ set. Finally, the $UPDSEED_i$ is incrementally calculated as the union of the $AFFECTED\_CORES_i$ set, and the set of all the potential and core microclusters that belong to the same offline cluster of any of the members of the $AFFECTED\_CORES_i$ set.

In Lines 11-19 of Algorithm 5.3, all the potential microclusters are found which are affected by removing the potential microclusters which faded out into outliers in the online part. For each potential microcluster $c_q \in \mathcal{N}_{\varepsilon_F}(c_d)$, the preference subspace vector of each c_q is updated and added to $AFFECTED\_CORES_d$ if the core member property of c_q has changed because

Algorithm 5.3: updateClustering(C)

for all $c_p \in Inserted\_PMC$ **do**
  compute the subspace preference vector w_{c_p}
  for all $c_q \in \mathcal{N}_{\varepsilon_F}(c_p)$ **do**
    update the subspace preference vector of c_q
    if core member property of c_q has changed **then**
      add c_q to $AFFECTED\_CORES_i$
    end if
  end for
  compute $UPDSEED_i$ based on $AFFECTED\_CORES_i$
end for
for all $c_d \in Deleted\_PMC$ **do**
  for all $c_q \in \mathcal{N}_{\varepsilon_F}(c_d)$ **do**
    update the subspace preference vector of c_q
    if core member property of c_q has changed **then**
      add c_q to $AFFECTED\_CORES_d$
    end if
  end for
  compute $UPDSEED_d$ based on $AFFECTED\_CORES_d$
end for
$UPDSEED \leftarrow UPDSEED_i \cup UPDSEED_d$
for all $c_p \in UPDSEED$ **do**
  remove the $ClusterID$ of old offline cluster that contained c_p, if any;
  assign c_p as unclassified;
end for
call *expandCluster(UPDSEED)*; *// cf. Algorithm 5.4*

of deleting c_d out of its ε_F-neighborhood. Then, similar to the insertion case, the $UPDSEED_d$ is incrementally calculated as the union of the $AFFECTED\_CORES_d$ set, and the set of all the potential and core microclusters that belong to the same offline cluster of any of the members of the $AFFECTED\_CORES_d$ set.

After that all the affected potential microclusters were found, then $UPDSEED_i$ and $UPDSEED_d$ can be merged to $UPDSEED$ as in Line 20. Then, in Lines 21-24, the $ClusterID$s of the affected old offline clusters of all potential microclusters of $UPDSEED$ are deleted to give a place for the new clustering result. Then all of these microclusters are assigned as *unclassified* to be reinserted correctly into the clustering. This is done by calling a modified version of the function *expandClusters()* of the algorithm PreDeCon [BKKK04] for all the potential microclusters in $UPDSEED$ in Line 25. The result of $expandCluster(UPDSEED)$ is that all the potential microclusters $c_p \in UPDSEED$ are either categorized into a cluster or marked as noise.

For the sake of completeness, Algorithm 5.4 details our modified version of *expandCluster()*. In a DBSCAN way, we pick a core microcluster from the $UPDSEED$ and insert its weighted ε_F-neighborhood in a queue Φ. Then we pick each members c_q from Φ to recursively insert in Φ all its directly reachable microclusters. And then iteratively expand the cluster.

5.5 Experimental Evaluation

In this section, the experimental evaluation of PreDeConStream is presented. PreDeConStream, as well as the two comparative algorithms: HPStream as a k-means-based projected algorithm and, DenStream as a full-space density-based algorithm, were implemented in Java. The experiments in Sections 5.5.3 and 5.5.3 were done on a Linux operating system with a 2.4 GHz processor and 3 GB memory. The parameter sensitivity experiments in Sections 5.5.3 and 5.5.3 were performed using the Subspace MOA subspace stream clustering environment [HKS13] (cf. Chapter 9) on a Windows system with a 3.0 GHz quad-core processor and a 4 GB memory.

Algorithm 5.4: expandCluster($UPDSEED$)

```
1: for all unclassified c_p ∈ UPDSEED do
2:   if CoreMC(c_p) then
3:     generate a new ClusterID;
4:     insert all c ∈ N_{ε_F}^{V_cp}(c_p) into queue Φ;
5:     while Φ ≠ ∅ do
6:       c_q = first microcluster in Φ;
7:       compute M = {c ∈ UPDSEED | DirReach(c_q, c)};
8:       for all c ∈ M do
9:         if c is unclassified then
10:          insert c into Φ;
11:        end if
12:        if c is unclassified or noise then
13:          assign current ClusterID to c;
14:        end if
15:      end for
16:      remove c_q from Φ;
17:    end while
18:  else
19:    mark c_p as noise;
20:  end if
21: end for
```

5.5.1 Datasets

For the evaluation of PreDeConStream several datasets were used:

1. *Synthetic Dataset: SynStream3D* consists of 3-dimensional 4000 objects without noise that form at the beginning two arbitrarily shaped clusters over full space (cf. Figure 5.2(a)). After some time, the data stream evolves so that for each cluster different dimensions of both clusters become irrelevant (cf. Figures 5.2(b) and 5.3).

2. *Synthetic Dataset: N100kC3D50R40* generated similar to the way mentioned in DenStream [CEQZ06] with 100000 data objects forming 3 clusters with 40 relevant dimensions out of 50.

3. *Synthetic Dataset: RBFSubSpaceGenerator* offered by Subspace MOA framework [HKS13] (cf. Chapter 9) as a synthetic random RBF subspace generator with the possibility of varying multiple parameters of the generated stream and its subspace events. One can vary: the number of dimensions, the number of relevant dimensions (i.e. the number of dimensions of the subspaces that contain the ground truth clusters), the number of the generated clusters, the radius

of the generated clusters, the speed of the movement of the generated clusters, the percentage of the allowed overlapping between clusters, and the percentage of noise. Note that some dimensions of a point could represent a noise within some subspace, while other dimensions could be a part of a ground truth cluster in other subspace. The generated noise percentage in this case represents a guaranteed noise in all subspaces. For this dataset, we had the following settings [HKS13] in our experiments: modelRandSeed=1, instaceRandomSeed=5, number of subspace clusters=5, Kernel Radius=0.07, noise level= 10%, the speed of movement of kernels=1% of the whole range every 200 points, the subspace clusters bounce when they reach the maximum range, additional event: deletion and creation of subspace clusters, frequency of the event= each 1000 points, decayHorizon=1000.

4. *Real Dataset: Network Intrusion Detection Dataset KDD CUP'99 (KDDcup)[Dat99]* explained in previous chapters, with 494021 TCP connections, each represents either a normal connection, or any of 22 different types of attacks (cf. Table 5.1). Each connection consists of 42 dimensions. In our experiments, all the 34 continuous attributes were considered.

5. *Real Dataset: Physiological Data, ICML'04 (PDMC) [Phy]* also explained in previous chapters is a collection of activities which was collected by test subjects with wearable sensors over several months. The dataset consists of 720792 data objects, each data object has 15 attributes and consists of 55 different labels for the activities and one additional label if no activity was recorded. Each object is labeled with the corresponding activity, 5102 for instance, stands for "sleeping", 3104 stands for "watching TV" and more. If no activity was recorded, the record is labeled as 0 which is also the dominant label in the dataset. In our experiments, 9 continuous attributes were considered after excluding the timestamp.

5.5.2 Evaluation measures and Parameter settings

To evaluate the efficiency of our algorithm, the runtime in seconds was measured. For the quality of the resulted clustering, four measures were used: the clustering purity [AHWY04, CEQZ06], the Entropy [SZ05], the F1 [AKMS08] and SubCMM [HKCS13] (cf. Chapter 10) measures were used. The clustering purity reflects how pure is each of the resulted clusters. The entropy, the F1 and SubCMM are newly used in this field. The F1 measure is well known from the static data, it mixes both the precision and the recall. We will explain the purity, the en-

tropy and the SubCMM in more details in the following sections, and a further explanation of SubCMM can be found in Chapter 10.

The purity as clustering quality evaluation measure

The purity measure is widely used [CEQZ06, AHWY04, KABS09, HSGS12] to evaluate the quality of a clustering. Intuitively, the purity can be seen as pureness of the final clusters compared to the classes of the ground truth. The average purity is defined as follows:

$$\text{purity} = \frac{\sum_{i=1}^{N} \frac{|C_i^d|}{|C_i|}}{N}$$

where N represents the number of clusters, $|C_i^d|$ denotes the number of objects with the dominant class label in cluster i and $|C_i|$ denotes the number of the objects in cluster i. The purity is only computed over a certain pre-defined window H from the current time. This is done since the weights of the objects decay over time.

The entropy as a projected stream clustering evaluation measure

The entropy [SZ05] measures the homogeneity of the found clusters with respect to the classes in the ground truth. For a clustering $\mathcal{C}$, the entropy is defined as:

$$E(\mathcal{C}) = -\sum_{C_j}(\frac{n_j}{n}\sum_{i} p_{ij} log(p_{ij}))$$

where n_j is the number of points in the C_j, the j-th cluster of $\mathcal{C}$, n is the total number of points, $p_{ij} = \frac{n_{ij}}{n}$ and n_{ij} is the number of points which are assigned to C_j while actually belong to the ground truth class i.

In the above formula, the lower the entropy the better the clustering. For a better readability and visualization of the results in our measurements, we have subtracted the above formula from 1 after normalizing it. Thus, the value 1 of the entropy in our results would mean a perfect clustering, while 0 will mean the worst result.

SubCMM (Subspace Cluster Mapping Measure) as a projected stream clustering evaluation measure

SubCMM [HKCS13] (Chapter 10) is an effective evaluation measure for stream subspace clustering that is able to handle errors caused by emerging, moving, or splitting subspace clusters. It is so far, the only subspace clustering measure that is dedicated to measure the quality of subspace clustering of streaming data. In a *d*-dimensional space, the measure first divides each object into *d* "sub-objects". The ground truth, should contain then the real subspace cluster, where each sub-object belongs to. The measure then consists of three steps:
First, each found subspace cluster is assigned to one of the ground truth subspace clusters based on class distribution in each cluster. Then, class frequencies are counted for each cluster, and each prediction cluster is mapped to a ground truth cluster that has the most similar class distribution.
Second, the penalty for every incorrectly predicted sub-object is calculated. The penalty is calculated according to the *connectivity* of the wrongly clustered sub-objects to their real subspace and their real cluster in that subspace. If a fault sub-object is closely connected to its hidden (ground truth) subspacce cluster, then the error becomes much severe since it was meant to be easily clustered to it. On the other hand, if the sub-object has high connectivity to the found subspace cluster, SubCMM allows a low penalty since it was hard to be clustered correctly. The same applied also to the connectivity to the subspace.
Third, derive a final SubCMM value by summing up all the penalties weighted over its own lifespan.

The measure ranges between 0 and 1, higher values imply better mapping of found subspace clusters to the ground truth. While in subspace clustering one object might be a part of multiple subspace clusters, using SubCMM is easier in projected clustering, where an object can belong to maximally one cluster.

Parameter Setting

In PreDeConStream the offline parameters ε_F and μ_F specify the density threshold that the microclusters must exceed in order to become core or potential (in the offline phase). A lower bound for ε_F is the online parameter ε_N. In the experiments, ε_F was set to at least $2 \times \varepsilon_N$. Unless otherwise mentioned, the parameters for PreDeConStream were set similar to [CEQZ06] as follows: decay factor $\lambda = 0.25$, initial data object $Init = 2000$, and horizon $H = 5$.

5.5.3 Experiments

To evaluate PreDeConStream, it is compared to the stream algorithms DenStream and HPStream and its sensitivity to some parameters was also checked. Section 5.5.3 shows the performed experiments to check the quality of the clustering results. Section 5.5.3 lists efficiency results of our algorithm. and Sections 5.5.3 and 5.5.3 show the parameter sensitivity of our algorithm against the parameter τ and the stream speed, respectively.

Evaluation of Clustering Quality

Using the SynStream3D dataset for both PreDeconStream and DenStream, the online parameters are set to $\varepsilon_N = 4$, $\mu_N = 5$ and $\lambda = 0.25$. For PreDeConStream, the maximal preference dimensionality is set to $\tau = 2$ and $\mu_F = 3$. The stream speed was set to 100 points per time unit. With this speed setting, the data stream evolves at timestamp 26, i.e. one dimension for each cluster becomes irrelevant. It can be seen from Figure 5.8(a) that the cluster purity of PreDeConStream and DenStream is 100% until the data stream changes, which is not the case for HPStream. This is because both can detect clusters with arbitrarily cluster shapes. Beginning from time unit: 26 the stream evolves such that in each cluster one dimension is no longer relevant and thus DenStream as a full-space clustering algorithm, does not detect any cluster. Similarly, Figure 5.8(b) shows the purity results of both algorithms over the N100C3D50R40 dataset. It can be seen that PreDeConStream outperforms HPStream. The time units are selected in such a way that the changes of the cluster purity can be observed when the stream evolves. It can be observed that HPStream has problems with detecting the changes in the stream. That is because the radius of the projected clusters might be too high and the new points are wrongly clustered. PreDeConStream adapts to the changes in the stream fast and keeps a high cluster purity.

We evaluated PreDeConStream against the state-of-the-art HDDStream algorithm [NZP+12] (cf. Section 5.2.3) using the RBFSubSpaceGenerator synthetic dataset. The evaluation of the results was happening after each 1000 points. We used a stream of 15000 points, and varied the dimensionality of the data to compare the reaction of both HDDStream and PreDeConStream to the change of the dimensionality of the data. The parameter settings for both algorithms were set equally wherever applies. Special parameters of HDDStream, were set as recommended in [NZP+12].

Normal or attack Type	Objects within horizon $H = 5$ at time unit 150	350	373	400
normal	4004	4097	892	406
satan	380	0	0	0
buffer overflow	7	1	2	0
teardrop	99	99	383	0
smurf	143	0	819	2988
ipsweep	52	182	0	0
loadmodule	6	0	0	1
rootkit	1	0	0	1
warezclient	307	0	0	0
multihop	0	0	0	0
neptune	0	618	2688	1603
pod	0	1	99	0
portsweep	0	1	117	1
land	0	1	0	0
sum	5000	5000	5000	5000

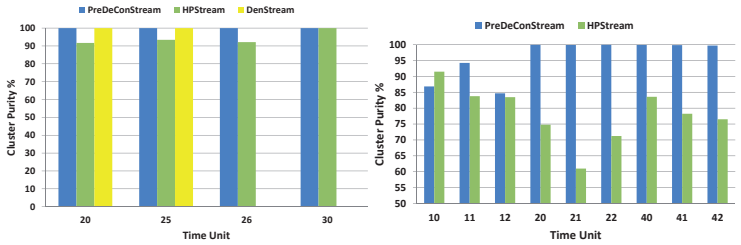
Table 5.1: A Table of the contents of the Network Intrusion Dataset [Dat99] as a stream data within a horizon $H = 5$ and with a stream speed $= 1000$

Dimensionality of the Data	2	3	4	5	6	7	8
Avg. Dim. of Sub. Clusters	1	2	3	3	4	5	6

Table 5.2: The average dimensionality of the subspace clusters with each data dimensionality in Figure 5.9(a)

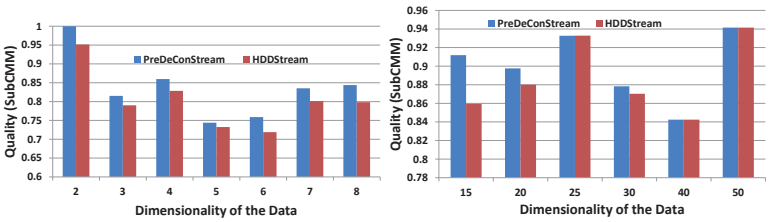
Figure 5.9(a) depicts the averaged SubCMM values over all the stream for lower dimensionalities and Table 5.2 shows the according average dimensionality of the generated subspace clusters. It can be seen that PreDeConStream can always map the subspace clusters in all of lower dimensionalities slightly better than HDDStream. The fact that both algorithms are building above PreDeCon [BKKK04] in their offline parts, makes their delivered results very similar. The advantage that PreDeConStream has over HDDStream in this context, is the faster online management of microclusters, which enables PreDeConStream to faster adapt to the changes of the evolving stream.

Figure 5.9(b) depicts the the averaged SubCMM values over all the stream for higher dimensionalities and Table 5.3 shows the according average dimensionality of the generated subspace clusters. Again, for higher dimensionalities, PreDeConStream has almost always the same clustering quality, and sometimes



(a) (b)

Figure 5.8: Clustering purity for: (a) SynStream3D dataset, (b) N100C3D50R40 dataset.



(a) (b)

Figure 5.9: Clustering quality using SubCMM for the *RBFSubSpaceGenerator* synthetic dataset for: (a) lower dimensionality, (b) higher dimensionality.

even better than HDDStream. The other advantage that PreDeConStream has over HDDStream is the fast and efficient production of offline subspace clusters, as we will see in Section 5.5.3. There, and for a fair comparison, we repeated the same settings as in Figure 5.9(b) when comparing the runtime of both algorithms.

On the Network Intrusion Dataset, the stream speed was set to 1000 points per time unit. Since the Network Intrusion dataset was already used in [AHWY04, CEQZ06], the same parameter settings are chosen for DenStream and HPStream as in [AHWY04, CEQZ06]. Since PreDeConStream also builds on a microcluster

Dimensionality of the Data	15	20	25	30	40	50
Avg. Dim. of Sub. Clusters	13	17	22	25	30	45

Table 5.3: The average dimensionality of the subspace clusters with each data dimensionality in Figures 5.9(b) and 5.13

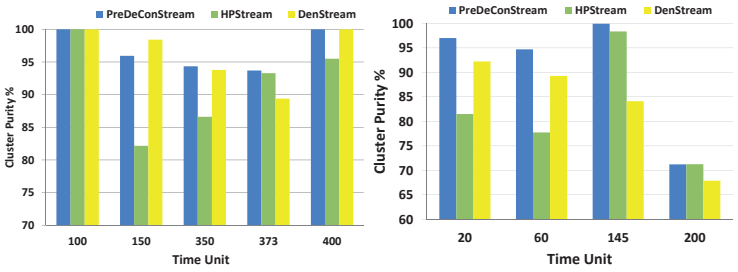


Figure 5.10: Clustering purity for: (a) KDDcup dataset, (b) PDMC dataset.

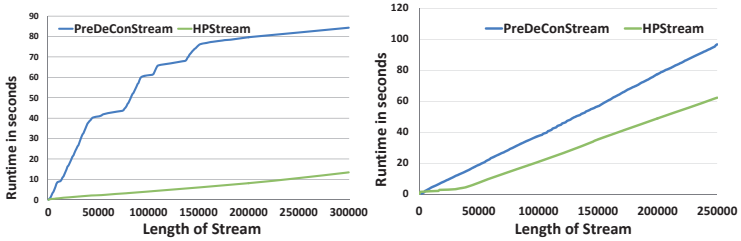
structure, similar parameter settings for the online part of PreDeConStream are chosen, to have a fair comparison. For PreDeConStream: $\beta = 0.23$, $\mu_F = 5$, and $\tau = 32$. For all the three algorithms, the decaying factor λ is set to 0.25.

Figure 5.10(a) shows the purity results for the KDDcup Dataset. It can be seen that PreDeConStream produces the best possible clustering quality. For the evaluation, measurements at the timestamps where some attacks exist, were selected. The data recordings at timestamp 100 and all the recordings within the horizon 5 were only attacks of the type "smurf". At this time unit any algorithm could achieve 100% purity. The attacks that appeared within horizon $H = 5$ in different timestamps are listed in Table 5.1. By comparing Table 5.1 and Figure 5.10(a), one can observe that PreDeConStream is also resistant against outliers. At the timestamp 350 and 400 there were some outlier attacks within the horizon which affected other algorithms less than PreDeConStream.

Figure 5.10(b) shows the purity results over the Physiological dataset. The stream speed $= 1000$ and $H = 1$. Again, the timestamps were selected in such a way that there are different activity labels within one time unit. It can be seen from Figure 5.10(b) that PreDeConStream has the highest purity.

Evaluation of Efficiency

The real datasets are used to test the efficiency of PreDeConStream against HPStream. The parameters were set the same way as for the previous experiments on these datasets and the results are shown in Figure 5.11. Although it is unfair to compare the runtime of a completely density-based approach against a *k*-means-based one, but Figures 5.11(a) and 5.11(b) show a considerable positive



(a) (b)

Figure 5.11: Runtime results for: (a) KDDcup dataset with a clustering request at each time unit, (b) PDMC dataset with a clustering request at each 20th time unit.

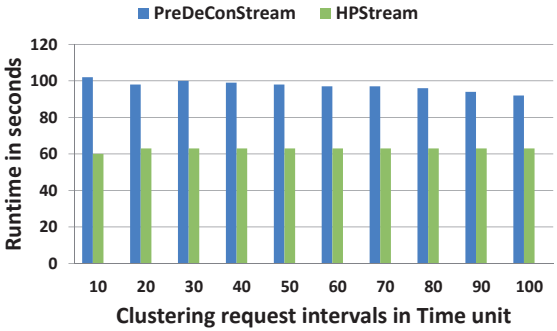


Figure 5.12: The runtime result for the PDMC dataset with different clustering request intervals.

effect of our clustering maintenance model when the frequency of the clustering requests decreases. Usually, clustering requests are not extremely performed at each timestamp or even at each 20th timestamp. This fact motivated a further experiment where we tested the performance of the two algorithms for different clustering frequencies. The result which is depicted in Figure 5.12 confirms our assumption. Due to its clustering maintenance model, the performance of PreDeConStream performs better with higher clustering requests intervals.

We compared the efficiency of PreDeConStream against HDDStream using the *RBFSubSpaceGenerator* synthetic dataset. For a fair comparison, we repeated the same settings as in Figure 5.9(b) to show compare also the quality results with the performance results of both algorithms. Figure 5.13 shows the total runtime

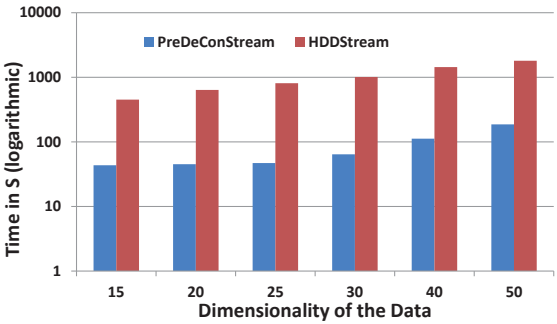
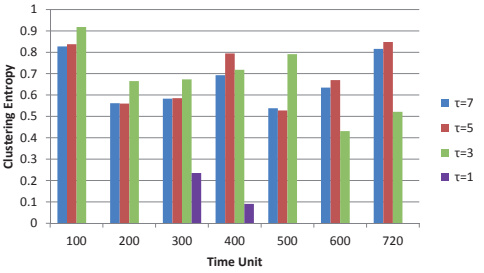


Figure 5.13: The runtime result for the *RBFSubSpaceGenerator* synthetic dataset with different dimensionalities of the data and subspace clusters (cf. Table 5.3).

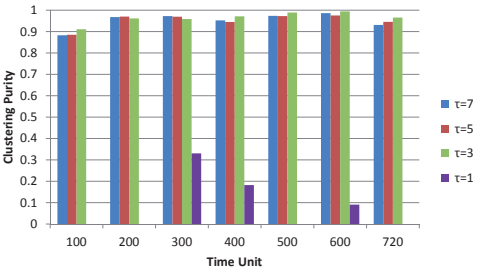
in Seconds of both algorithms. Note that this runtime includes also the runtime of underlying the Subspace MOA framework which includes a visualization interface, a calculation of two evaluation measures and intermediate outputting and plotting of the measures values. Of course, the same settings were applied for both algorithms. Please mind the logarithmic scale of the second in runtime axis in Figure 5.13. The superiority of PreDeConStream is obvious in the results. For all dimensionalaities of the dataset, PreDeConStream is faster than HDDStream by more than an order of magnitude. There are tow main reasons for the high speed of PreDeConStream: 1. The lower complexity of the online phase, since HDDStream summarizes both the points and their dimensionality into the microclusters in a heavy, PreDeCon-similar way. 2. The high complexity of the offline part in HDDStream, which is the dominating one, is optimized in PreDeConStream by the continuous updating of the previous solutions, and updating only the areas which where affected since the last offline output. It can be seen also from Figure 5.13 that HDDStream is much more sensitiv to increasing the dimensionality of the data than PreDeConStream.

Varying τ: the number of maximal irrelevant dimensions

The parameter τ in our algorithm represents the maximal number of dimensions that are allowed to be irrelevant while detecting the core microclusters in the offline phase. Thus, it is the key parameter of the idea of projected clustering, since it reflects how powerful the projected clustering will be when compared to the full-space one, according to the clustering quality. We have used for that



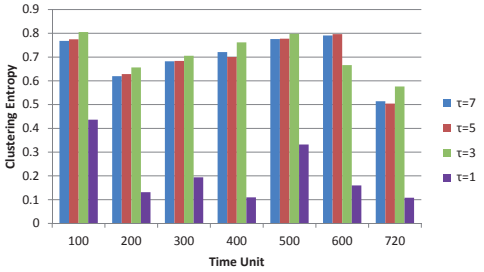
(a)



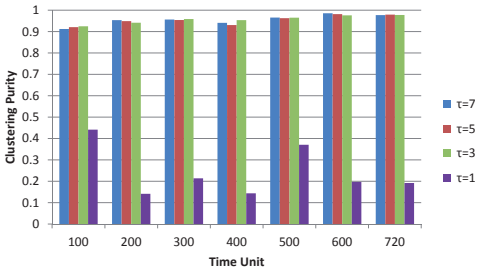
(b)

Figure 5.14: Clustering quality for different values of τ: the maximum number allowed number of irrelevant dimensions, using the PDMC dataset, with **H=10**: (a) Clustering entropy, (b) Clustering purity.

testing purpose the PDMC dataset with its 9 continuous dimensions. For the parameter settings, we selected $\varepsilon_N = 2$, $\varepsilon_F = 4$, $\beta = 0.3$, $\mu_N = 5$, $\mu_F = 5$, $\lambda = 0.25$, $Init = 2000$, $\kappa = 10$, $\delta = 0.1$ and Stream Speed$= 1000$. The value $\tau = 1$ represents a strict case that allows the clusters with only one dimension to be irrelevant, which is close to the full-space clustering. While a value $\tau = 7$ represents a very flexible case that accepts clusters with only 2 out of the 9 dimensions of the dataset to be relevant. Figure 5.14 depicts the results when considering the horizon $H = 10$. Both the entropy and the purity of the resulted clusterings are extremely low when $\tau = 1$. This case is very similar to the full-space clustering algorithms (where "$\tau = 0$"). Such strict algorithms do not deliver clusters with many irrelevant dimensions, although they are existing in the ground truth. Thus, the purity of the formed clusters falls down, and the homogeneity between their members degrades. It can also be seen, that clustering quality is maximized when selecting $\tau = 3$ and $\tau = 5$. Figure 5.15 shows the same results but with a



(a)



(b)

Figure 5.15: Clustering quality for different values of τ: the maximum number allowed number of irrelevant dimensions, using the PDMC dataset, with **H=100**: (a) Clustering entropy, (b) Clustering purity.

higher horizon $H = 100$ for more information about a longer window over the same settings.

Varying the stream speed

The stream speed represents another important issue of stream clustering algorithms. In subspace and projected stream clustering, it makes an additional challenge for the algorithm. High stream speeds make it more difficult for the projected stream clustering algorithm to detect clusters within all the subspaces of the evolving stream. For these experiments, we have also used the PDMC dataset, and the same parameter settings as in Section 5.5.3, with $\tau = 7$ and $H = 100$. We varied the stream speed as: 50, 10, 200 and 1000. Figure 5.16(a) depicts the purity results. Apparently, the results are against the intuition. For low stream speed (50) we have lower clustering purity, while higher purities are gained with

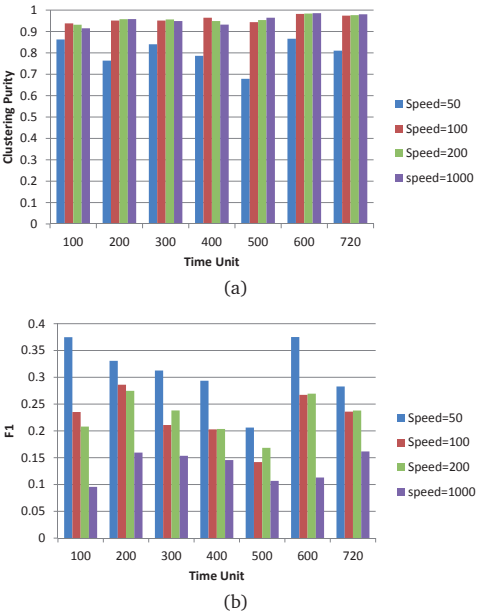


Figure 5.16: Clustering quality for different stream speeds using the PDMC dataset: (a) Clustering purity, (b) The F1 measure.

higher speeds. To double check these results, we get the F1 measures of the same settings (cf. Figure 5.16(b)). The F1 results assure our intuition. Slower stream speeds have higher precision values, and thus higher F1 amounts. Slower stream speeds gives more budget of time for the heavy projected clustering calculations, as well as the online maintenance of microclusters. Thus, these summaries will stay always up-to-date and will follow the evolution of the stream. As the stream speed increases these amounts decrease although the purity remains high.

5.6 Conclusion

In this chapter we have introduced a novel projected stream clustering algorithm termed *PreDeConStream*. Based on the two-phase process of mining data streams, our technique builds a microcluster-based structure to store an online summary of the streaming data. The technique is based on the subspace clustering concept

and targets applications with high dimensionality of complex streaming data. Localizing the change of the previous clustering result, in addition to the smart usage of the clustering validity interval made our model efficient in its offline phase. As a result, our technique has proved experimentally its superiority over state-of-the-art techniques.

Chapter 6

Hierarchical Adaptive Density-based Stream Clustering

* Due to the continuously evolving nature of streaming data, it is crucial that the algorithm autonomously detects clusters of arbitrary shape, with different densities, and varying number of clusters. Although available density-based stream clustering are able to detect clusters with arbitrary shapes and varying numbers, they fail to adapt their thresholds to detect clusters with different densities. In this chapter we propose a stream clustering algorithm *HASTREAM*, a **H**ierarchical, density-based, **A**daptive **STREAM** clustering algorithm. Our algorithm automatically detects evolving clusters of different densities. The density thresholds are independently adapted to the existing data without the need of any user intervention. To reduce the high computational cost of the presented approach, techniques from the graph theory domain are utilized to devise an incremental update of the underlying model.

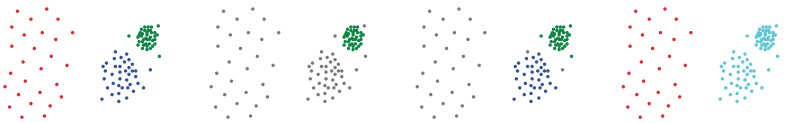
To show the effectiveness of HASTREAM and hierarchical density-based approaches in general, several synthetic and real world datasets are evaluated using various quality measures. The results showed that the hierarchical property of the model was able to improve the quality of density-based stream clusterings and enabled HASTREAM to detect streaming clusters of different densities.

*Parts of this chapter have been published in the Proceedings of the 10th International Conference on Machine Learning and Data Mining (MLDM 2014) [HSS14].

6.1 Motivation

Streaming data differs from a static dataset in various ways (cf. the discussion about challenges in sensor streaming data in Section 2.1.1). One of the main differences is that the data of a stream arrives continuously and has to be processed at its arrival. Furthermore, due to the huge amount of data, it can not be stored persistently. Another implication of the endless character of the stream is the impossibility of performing multiple passes over the data, and thus every new object can only be processed once. Every object has to be processed as fast as possible, thus the processing has to be efficient. An additional challenge is the evolution of data streams, thus the algorithm has to recognize and *continuously* adapt to these changes in the data.

Most stream clustering algorithms follow the online-offline-phases model. In the online phase, a summary of the evolving data is performed and continuously maintained to follow the changing distribution of the data. The offline phase is then performed upon a user request, and uses one of the well known static clustering algorithms to deliver the final clustering. CluStream [AHWY03] for example, performs a k-means variant in the offline phase over the data summaries (also called microclusters in most algorithms). Motivated by the need to detect arbitrarily-shaped evolving clusters, and being less sensitive to outliers in the stream, density-based stream algorithms were developed like DenStream [CEQZ06] which obtains the final clustering by performing a DBSCAN variant over the microclusters. The benefits of density-based clustering algorithms are that they are able to find clusters of arbitrary shape and autonomously detect the number of clusters. However a significant drawback of available density-based approaches, is that they are not able to find clusters of different densities, due to the "static" density threshold parameter that is usually used in such algorithms (cf. Figure (6.1)). Having clusters of different densities at the same time is very common in streaming data. Additionally, the same clusters may evolve over the stream such that they will have different densities over the time. Setting the density threshold parameter to a single value all over the stream (like e.g. in DenStream), will lead to missing many clusters (cf. Figure (6.1)). One approach to counteract this problem is to use hierarchical clustering techniques to enable density-based clustering algorithms to find clusters of different densities, by adapting the density threshold to the characteristics of each considered cluster.



(a) Ground Truth (b) $\varepsilon = 5$ (c) $\varepsilon = 12$ (d) $\varepsilon = 30$

Figure 6.1: Clusters can not be all detected with a single density threshold as in DBSCAN [EKSX96]. (b) If ε is too small , only the green cluster is found. However, if ε is chosen too high (d), the blue and green cluster are detected as a single cluster. [$\mathrm{minPts} = 5$]

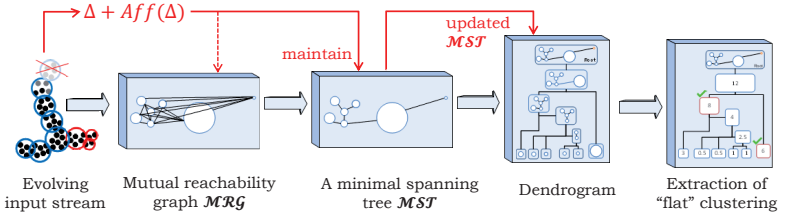


Figure 6.2: The general concept of HASTREAM algorithm. The red arrows represent the incremental contribution.

In this chapter, we propose, the first hierarchical density-based stream clustering algorithm based on cluster stability, called *HASTREAM*. The algorithm is able to detect clusters of different densities and arbitrary shapes. The final *flat* clustering is extracted from the hierarchical clustering. The density parameters as well as the number of clusters are automatically adapted while extracting the clustering, without any need for user intervention. Figure (6.2) presents an overview of the HASTREAM algorithm as partially adopted from [CMS13]. Since the contribution of HASTREAM concentrates on the offline phase, its input are the current summaries, called microclusters, of the evolving stream. The algorithm first build a complete graph, called the mutual reachability graph $\mathcal{MRG}$, out of the current microclusters. Then, a minimal spanning tree $\mathcal{MST}$ is extracted out of the $\mathcal{MRG}$. According to the weights of the microclusters and the distances between them in $\mathcal{MST}$, a dendrogram is built in the next step to reflect the hierarchical clustering. A novel weighted cluster stability measure extracts the most stable flat clustering in the final step. This clustering adapts to the distribution of the stream without any need to a user intervention.

The remainder of this chapter is presented as follows: Section 6.2 gives an

overview on the related work from different relevant areas. Section 6.3 presents some needed definitions and the used data structures in addition to the introduced graph maintenance novel method and the stability measure for extracting the final clustering. In Section 6.4, we present our HASTREAM algorithm in details. In Section 6.5, we thoroughly evaluate two variants of *HASTREAM* against a state-of-the-art algorithm before concluding the chapter in Section 6.6.

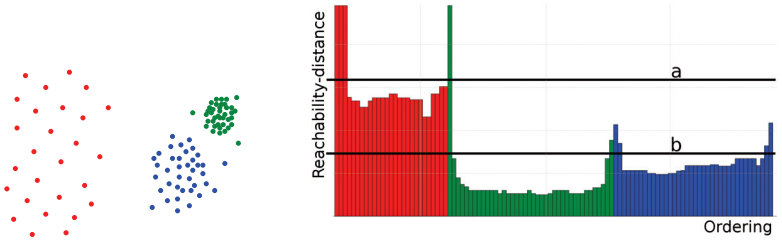
6.2 Related Work

In this section, a detailed review of literature related to hierarchical clustering of static data is presented. Furthermore, this chapter presents an overview of the well known streaming algorithms. Additionally, a short introduction in graph-based clustering is given, as it is a prerequisite to our algorithm.

6.2.1 Hierarchical Clustering

DBSCAN [EKSX96] has been the most appealing density-based clustering algorithm for a long time. A known problem in DBSCAN however, is detecting clusters with considerably different densities (cf. Figure (6.1)). Hierarchical clustering [ABKS99, HKP06, ZJ14] is a clustering technique with the objective of decomposing the data set into a hierarchy of clusters. Hierarchical-based clustering algorithms are able to find clusters of different densities and nested clusters. There exist two basic approaches called agglomerative and divisive hierarchical clustering. Agglomerative methods use the bottom-up and divisive methods use the top-down approach. Each object is represented as a cluster at the beginning. These clusters are merged into larger clusters until only one cluster remains. Divisive methods, on the other hand, start with all the objects in the same cluster and split the clusters until each object is its own cluster. In both approaches, the clusters are either merged or split according to a dissimilarity measure. The distance between the clusters can be determined in different ways. Commonly used distance functions are single-linkage, complete-linkage and average-linkage. Single-linkage determines the minimal distance between the two nearest objects in the clusters. Complete-linkage determines the maximal distance between the objects in the clusters. Average- linkage determines all the pairwise distances between the objects in the clusters and takes the average. **OPTICS:** Ordering Points To Identify the Clustering Structure [ABKS99] requires two parameters ε

and $minPts$, which have the same meaning as in DBSCAN. OPTICS orders the objects such that the closest objects are also neighbors in the ordering. To perform this ordering, the algorithm computes two additional values for each object, called core-distance and reachability-distance. The reachability-distance is crucial for extracting the final clustering. The final cluster ordering can be visualized



(a) Ground Truth (b) Reachability plot

Figure 6.3: The two lines in the reachability plot (b) represent two different thresholds. Using the upper line, representing a higher threshold, leads to two clusters, where the blue and green areas are recognized as a single cluster and the red cluster is correctly recognized. The lower line, corresponding to a lower threshold, also recognizes the blue and green clusters.

by the reachability-plot. Figure (6.3) illustrates a reachability-plot for a 2D dataset. The horizontal axis represents the ordering of the objects and the vertical axis their respective reachability-distance. The clusters can be identified by the valleys in the reachability-plot. By setting a threshold, represented by a horizontal line, the clustering can be extracted from the reachability-plot. Each valley the horizontal line crosses represents a cluster. **HDBSCAN:** Density-Based Clustering Based on Hierarchical Density Estimates [CMS13]. Is a recent research motivated by the fact that OPTICS only allows a flat clustering by setting a single density threshold. Thus, OPTICS may not lead to the optimal clustering if the clusters have different densities, as illustrated in Figure (6.3). Campello et al. [CMS13] proposed the density-based hierarchical clustering algorithm HDBSCAN. The authors introduced a new cluster stability measure to extract a flat clustering, that contains the most significant clusters from the hierarchy of clusters delivered by HDBSCAN. HDBSCAN builds the hierarchical clustering by computing the $minPts$-core-distance for each object of the dataset. Then, a complete graph (called mutual reachability graph) is computed where each object is represented

by a vertex and the weight of an edge is equivalent to the mutual reachability distance of the two considered objects. A minimal spanning tree can be extracted from the mutual reachability graph. This minimal spanning tree is used to build a dendrogram that represents the hierarchical clustering, by iteratively removing the largest weighted edge. The newly introduced measure *cluster stability* is used to extract a flat clustering from the dendrogram. The cluster stability measure is an adaptation of the excess of mass concept [MS91]. The cluster stability of a cluster is directly related to the density of the cluster. Higher densities infer more stability. Thus the extracted (flat) clusters should have the highest possible overall summed cluster stability. The cluster stability of each cluster is computed after constructing the hierarchical clustering. It should be noted that when the flat clustering is extracted, it is prohibited to select nested clusters. The cluster stability can be used to decide whether to choose a cluster at a higher hierarchy level or a combination of non-overlapping subclusters on a lower hierarchy level. A more detailed explanation of the cluster stability measure is available in [CMS13].

6.2.2 Graph-Based Clustering

Graph-based clustering techniques for static data is a well-established research area [AW10]. A popular method is the concept of minimal spanning trees. Given a connected and undirected graph, a Minimal Spanning Tree (MST) is a subgraph which connects all the vertices and minimizes the sum of the edge costs. Furthermore a MST does not contain any cycles. Minimal spanning tree clustering techniques are used in various algorithms [WWW12, GZJ06] as well as the presented algorithm HDBSCAN [CMS13]. However one of the downsides of algorithms based on minimal spanning trees is that they have a considerably high computational complexity. Thus considering MST-based clustering techniques in the data stream scenario is not straight forward. The following section presents algorithms for constructing minimal spanning trees and their computational complexities. Furthermore a short overview for maintaining minimal spanning trees is given.

6.2.3 Construction of a Minimal Spanning Tree

Let $\mathcal{G}(V, E)$ be a graph with set of vertices V and set of edges E. The complexity of building an MST from graph $\mathcal{G}$ depends on the used approach. The most well

Algorithm	Runtime Complexity	
	Standard	Fibonacci Heap
Borůka	$O(\|E\| \log \|V\|)$	-
Prim	$O(\|V\|^2)$	$O(\|E\| + \|V\| \log \|V\|)$
Kruskal	$O(\|E\| \log \|V\|)$	-
Dijkstra	$O(\|V\|^2)$	$O(\|E\| + \|V\| \log \|V\|)$
Karger	expected $O(\|E\|)$	-
Chazelle	$O(\|E\|\alpha(\|E\|, \|V\|))$	-

Table 6.1: Runtime complexity for building a minimal spanning tree

known approaches for building the MST are Borůvka's algorithm [Bor26], Prim's algorithm [Pri57], Kruskal's algorithm [Kru56] and Dijkstra's algorithm [Dij59]. Karger et al. [KKT95] proposed a randomized algorithm for building the MST with a linear expected complexity. Chazelle [Cha00] found a deterministic algorithm for building an MST with a complexity $O(|E|\alpha(|E|, |V|))$, where α is the inverse Ackerman function. Table (6.1) shows a complete overview of the computational complexities of the mentioned algorithms. The only method which is explained in the following is the Prim's algorithm since it is relevant for this chapter, while the remaining algorithms were mentioned for completeness. The Prim's algorithm starts with an arbitrary vertex $v \in V$. Then Prim's algorithm proceeds by augmenting the MST with the edge $e \in E$ that has the lowest cost and is adjacent to a vertex $w \in V$ that is not yet contained in the MST. If such an edge e exists, the adjacent vertex v is also incorporated into the MST. This process is repeated until every vertex of $\mathcal{G}(V, E)$ is contained in the MST. As shown in Table 6.1, the runtime of Prim's algorithm using an adjacency matrix is $O(|V|^2)$. For the streaming case reducing the runtime complexity is a main objective to be able to handle the huge amount of data. Fortunately, Prim's algorithm can be improved to a runtime complexity of $O(|E| + |V| \log |V|)$, by using a the Fibonacci heap [FT87]. A Fibonacci heap is a data structure that is composed of a collection of trees that satisfy the minimum-heap property. The advantage of using a Fibonacci heap is that the runtime complexity can be reduced significantly compared to using an adjacency matrix. This is especially the case when the graph density is high, i.e. the number of edges dominates the number of vertices.

6.2.4 The Maintenance of a Minimal Spanning Tree

In the literature, several methods exist, which are related to maintaining an existing MST. The maintenance of the MST is not a straightforward task as it needs

to consider the insertion of a vertex, the deletion of a vertex, the insertion of an edge and the deletion of an edge. Chin et al. [CH78] presented methods for updating the MST for the insertion and deletion of vertices. They showed that the insertion of a new vertex can be done in linear time. On the other hand, the proposed deletion method of a vertex has quadratic complexity with respect to the number of vertices. Furthermore the authors show that the insertion and the deletion of edges can be realized under the same runtime complexity constraints. The deletion case is the more complex task and, to the best of our knowledge, no existing approach can solve the problem in a linear time. Das et al. [DL01] used the concept of supervertices to improve the runtime complexity for deleting a vertex when maintaining an MST. A supervertex is a set of vertices, representing a connected component of an MST. The time complexity of the proposed algorithm is $O(|E|log|V|)$. Nardelli et al. [NPW04] proposed an nearly linear time algorithm for reconstructing the MST of a communication-based network after a node failure, which can be regarded as deleting a vertex. The time complexity is $O(|E|\alpha(|E|,|V|))$, where α is the inverse Ackerman function.

6.2.5 Data Stream Clustering Algorithms

A common approach for clustering streaming data is having two phases, an online and an offline phase. The online phase is responsible of summarizing the data stream, whereas the offline phase is responsible of the final clustering. The following algorithms use the mentioned approach. **CluStream** [AHWY03] summarizes the data stream by using the *microcluster* structure (Definition (5.1)). The offline component of CluStream considers the microclusters as pseudo-points and performs a k-means variant over the maintained microclusters. **DenStream** [CEQZ06] uses the microcluster structure as well. DenStream maintains two lists of microclusters, the potential and the outlier microclusters. The offline phase is a DBSCAN variant which considers only potential microclusters that exceed a certain density threshold. **ClusTree** [KABS09] (cf. also Chapter 8), is an anytime stream clustering algorithm which provides a compact and self-adaptive indexing structure for the microclusters to allow handling of fast and slow data streams. Therefore, the authors additionally propose new descent strategies to improve the clustering results. **OpticsStream** [TRA07] is a visualization algorithm which uses a microcluster structure to represent the data stream and the offline phase uses the object ordering technique of OPTICS [ABKS99] to generate

a 3-dimensional surface plot, called reachability surface. The idea is similar to the 2-dimensional reachability-plot of OPTICS, but the time dimension is added. The plot can be used to obtain insight on how the clusters change over time. **MR-Stream** [WND+09] uses a tree structure to represent the data stream. The data space is partitioned into equally sized cells. At each level of the tree structure, each cell is divided into subcells. E.g. if the number of cell divisions is set to 2, the resulting tree structure is similar to a quadtree. The offline clustering is performed on the cells at a user defined hierarchy level of the tree, by clustering all reachable dense cells to a cluster. Thus the algorithm does not solve the main drawback of density-based stream clustering algorithms, as it introduces a new parameter. Additionally it suffers from the efficiency issues that grid-based stream clustering algorithms usually have.

6.3 Preliminaries and Data Structure

Our algorithm, called HASTREAM, is composed of an online and an offline phase.

6.3.1 The Online Phase: microclusters List and Index Structures

The online phase uses the microcluster structure and the decaying mechanism that were introduced in Section 5.3.1 through Definitions (5.1 - 5.3). Only the microcluster maintenance slightly differs here. If an object p fits into a microcluster *mc*, the statistics are updated as follows:

$$mc = \left(\overline{CF^1} + p,\ \ \overline{CF^2} + p^2,\ \ w + 1\right)$$

Otherwise, if no object is added to the microcluster for a certain time interval δt, then it is updated as:

$$mc = \left(f_b(\delta t) \cdot \overline{CF^1},\ \ f_b(\delta t) \cdot \overline{CF^2},\ \ f_b(\delta t) \cdot w\right)$$

The offline part of the algorithm is built in an abstract way, such that the underlying microcluster structure can be easily exchanged. The two used microcluster structures are adapted variants of those used by DenStream [CEQZ06] (list structure) and ClusTree [KABS09] (index structure).

6.3.2 The Offline Phase: Density-Based Hierarchical Clustering

This model requires the density threshold parameter $minPts$ merely. To be able to explain the model, the required definitions are presented in the following. A microcluster is represented by a vertex when graphs are considered. Thus, when referring to the weight of a vertex, what is in fact referred to is the weight of the microcluster represented by the respective vertex.

Definition 6.1 (Core Distance) *The core distance of a microcluster mc_p is defined as the distance to its $minPts$-nearest neighbor microcluster.*

$$\text{core-dist}_{\text{minPts}}(mc_p) = \textit{distance to the } minPts\textit{-th closest microcluster}$$

Definition 6.2 (Mutual Reachability Distance) *The mutual reachability distance between two microclusters mc_p and mc_q is defined as the maximum of the Euclidean distance between the centers of the microclusters and their respective core-distances.*

$$dist_{mr}(mc_p, mc_q) = \max\{dist_2(mc_p, mc_q), \text{core-dist}_{\text{minPts}}(mc_p), \text{core-dist}_{\text{minPts}}(mc_q)\}$$

Definition 6.3 (Mutual Reachability Graph) *The mutual reachability graph $\mathcal{MRG}(V, E)$ is a complete graph. The set of vertices V is represented by the microclusters available at timestamp t. The weight of an edge from the set of edges E represents the mutual reachability distance between the two microclusters represented by vertices $u, v \in V$.*

$$\mathcal{MRG}(V, E) = \begin{cases} V = \textit{the set of available microclusters at time } t \\ E = \{e(u, v) \mid u, v \in V \textit{ with weight } w(e) = dist_{mr}(u, v)\} \end{cases}$$

Definition 6.4 (Adjacency List) *Given a connected graph $\mathcal{G}(V, E)$, the adjacency list of a vertex v contains all vertices which are directly reachable from v by one edge.*

$$\text{Adj}_E(v) = \{u \mid u \in V \textit{ and } e(v, u) \in E\}$$

Definition 6.5 (Connected Component) *A connected component is a set of vertices V, where each vertex has at least one adjacent vertex, for $|V| \geq 2$.*

$$\text{Conn-Component}(V, E) = \left\{ v \in V \mid \text{Adj}_E(v) \neq \varnothing \wedge \text{Adj}_E(v) \neq \{v\} \right\}$$

A single vertex is also a connected component.

Definition 6.6 (Total Weight of a Connected Component) *The total weight of a connected component* $\text{Conn-Component}(V, E)$ *is the sum of the weights of each* $v \in V$.

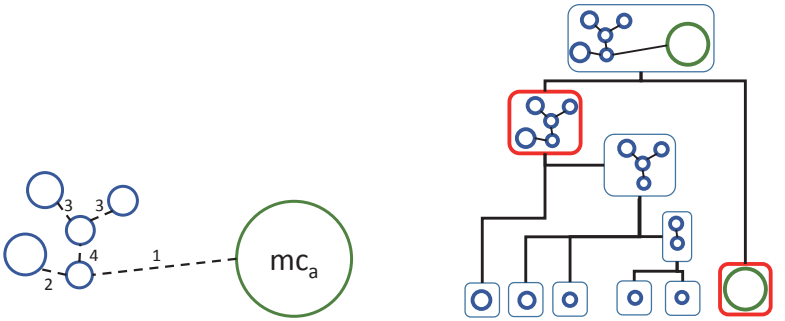
$$\textit{Total Weight}(\text{Conn-Component}(V, E)) = \sum_{v \in V} \textit{weight of } v$$

Definition 6.7 (Minimal Spanning Tree) *Given a connected and undirected $\mathcal{G}(V, E)$, a minimal spanning tree $\mathcal{MST}(V, T)$ is a connected subgraph of $\mathcal{G}$ which contains no cycles and minimizes the total weight w.r.t. the edges.*

$$\mathcal{MST}(V', T) = \begin{cases} V' = V \\ T = \left\{ e(u, v) \in E \;\middle|\; \textit{minimize } \sum_{e \in E} w(e) \right\} \\ \text{Conn-Component}(V', T) \end{cases}$$

Extraction of the Hierarchical Clusters

HDBSCAN [CMS13] wont be able to extract hierarchical clustering out of the microclusters when simply considering them as pseudo points. Figure (6.4(a)) illustrates the problem that could appear. If the extraction of the hierarchical clustering is only based on the centers of the microclusters, the microcluster mc_a would not be recognized as a cluster, even if the microcluster covers much of the relevant data. In this case, HDBSCAN would simply consider mc_a as noise, because it is far away and does not belong to the most stable clustering, even though the microcluster could form a cluster by itself. Thus, the hierarchical extraction model for the streaming case has to consider the weights of the microclusters. The weight of a microcluster is compared with the density threshold $minPts$. Following this approach, the microcluster mc_a from the example would



(a) Minimal spanning tree

(b) Dendrogram and extracted flat clustering (red boxes)

Figure 6.4: Minimal spanning tree over microclusters, the labels of the edges indicate the sequence in which the corresponding edges are removed.

be recognized as a cluster by itself, since its weight exceeds the density threshold.

The model for extracting the hierarchical clusters works as follows: Given a set of microclusters *MC*, the mutual reachability graph $\mathcal{MRG}$ is computed as defined in Definition (6.3) and the minimal spanning tree $\mathcal{MST}$ of $\mathcal{MRG}$ is extracted. Since the model is for streaming data, the extraction of the minimal spanning tree is a crucial step for the efficiency of the later algorithm. Thus in this model, the minimal spanning tree is built by using Prim's algorithm [Pri57] with a Fibonacci heap structure.

After computing the minimal spanning tree, the extraction of the clusters can begin. At the start, the root node of the dendrogram contains all microclusters, forming one cluster. Starting from a complete minimal spanning tree $\mathcal{MST}$, the largest edge from the $\mathcal{MST}$ is removed. If there exists multiple edges with the same weight, they have to be removed simultaneously. Removing the edges leads to connected subcomponents of the currently considered cluster. A subcomponent can be either a connected component with more than one microcluster, or it is a single microcluster. In case of a connected subcomponent, the total weight of the component is compared against $minPts$ and if it exceeds this density threshold, the subcomponent can be considered as a cluster at the current hierarchy level. Otherwise, the subcomponent is rejected and is no longer considered as a possible cluster, since the subcomponent can not form a cluster w.r.t. the density

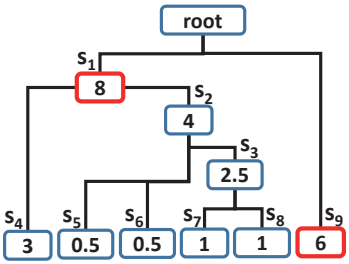


Figure 6.5: This figure illustrates a flat clustering extraction w.r.t. the cluster stability measure. The red boxes are selected, i.e. $s_1 = s_9 = 1$. Whereas, the blue boxes are not selected, i.e. $s_j = 0, \forall j \notin \{1, 9\}$.

threshold. In case of a single vertex, the weight of the microcluster, which is represented by the vertex , is compared against $minPts$ and is considered as a possible cluster when it exceeds the density threshold. Otherwise if its weight is lower than the density threshold, the microcluster can not form a cluster on its own.

This procedure is repeated iteratively until there is no edge left which can be removed. At the end, the hierarchical clustering is represented as a dendrogram. Figure (6.4(b)) illustrates the hierarchical clustering in the form of a dendrogram of the previous example. This model generates all the clusters which exceed the minimal density threshold $minPts$. This is because we filter the subcomponents at each step using the minimal density threshold.

Extraction of the Flat Clustering

In most applications, it is desired to have a flat clustering instead of returning every possible hierarchical cluster. For example in OPTICS [ABKS99], the flat clustering is obtained by setting a threshold in the reachability-plot. The resulting clustering is not necessarily optimal. In Section (6.2.1), the optimal clustering of the example dataset, cf. Figure (6.3), could not be found by a single threshold. In HDBSCAN [CMS13], a measure was presented to extract the optimal non-hierarchical clustering out of the hierarchical clusters. The referred measure, Cluster Stability, was already explained in more detail in Section (6.2.1). An example of a flat clustering extracted from the hierarchical clustering is illustrated in Figure (6.4(b)) and Figure (6.5). The measure for extracting the flat

clustering in this model is an adapted variant of the cluster stability measure for microclusters. The adapted cluster stability takes the weight of the microclusters into account and is defined as follows:

Definition 6.8 (Cluster Stability for Microclusters) *The cluster stability CS of a cluster $\boldsymbol{C}_i$ consisting of microclusters, is*

$$\begin{aligned} CS(\boldsymbol{C}_i) &= \sum_{mc_j \in \boldsymbol{C}_i} w(mc_j) \cdot \Big(\lambda_{max}(mc_j, \boldsymbol{C}_i) - \lambda_{min}(\boldsymbol{C}_i)\Big) \\ &= \sum_{mc_j \in \boldsymbol{C}_i} w(mc_j) \cdot \Big(\frac{1}{\varepsilon_{min}(mc_j, \boldsymbol{C}_i)} - \frac{1}{\varepsilon_{max}(\boldsymbol{C}_i)}\Big) \end{aligned} \tag{6.1}$$

where $w(mc_j)$ is the weight of the microcluster mc_j. $\lambda_{min}(\boldsymbol{C}_i)$ is the minimum density threshold at which $\boldsymbol{C}_i$ exists and $\lambda_{max}(mc_j, \boldsymbol{C}_i)$ is the density threshold where mc_j does no longer belong to the cluster $\boldsymbol{C}_i$. ε_{max} and ε_{min} are the corresponding ε thresholds.

ε_{max} and ε_{min} thresholds can be extracted for each cluster from the dendrogram.

Let $HC = \{\mathbf{C}_2, \ldots, \mathbf{C}_k\}$ be the set of all extracted hierarchical clusters from the dendrogram, except for the cluster $\mathbf{C}_1$ at the root node. In the following, it is assumed that the number of available microclusters is at least $minPts$ and thus the cluster $\mathbf{C}_1$ which contains all microclusters is not relevant. A flat clustering is a non overlapping set of clusters from the hierarchical clustering such that the most significant clusters are represented in the flat clustering. The most significant clusters are selected w.r.t. the highest cluster stability measure. The cluster selection can be formalized as an optimization problem. The goal is to maximize the sum of stabilities of the extracted clusters. The optimization problem of extracting the flat clustering is defined as follows:

Definition 6.9 (Flat Clustering Extraction) *Given a hierarchical clustering $HC = \{\boldsymbol{C}_2, \ldots, \boldsymbol{C}_k\}$, the flat clustering is the set of clusters which maximizes the sum of cluster stabilities.*

$$\begin{aligned} &\max_{s_2,\ldots,s_k} \quad \sum_{i=2}^{k} s_i \cdot CS(\boldsymbol{C}_i) \\ &\textit{with constraints} \begin{cases} s_i \in \{0,1\}, \quad 2 \le i \le k \\ \textit{if } (s_i = 1) \Leftrightarrow (s_j = 0) \textit{ , for all } j \in \mathcal{N}_i \end{cases} \end{aligned} \tag{6.2}$$

where $\mathcal{N}_i$ is the set of all the indices from the nested clusters of cluster $\boldsymbol{C}_i$.

The first constraint guarantees that a cluster $\mathbf{C}_i$ is either selected $(s_i = 1)$ or not selected $(s_i = 0)$ for the flat clustering. The second constraint guarantees that no nested cluster is contained in the flat clustering, i.e. if a cluster $\mathbf{C}_i$ is selected $(s_i = 1)$, all the nested clusters of $\mathbf{C}_i$ are not contained in the flat clustering.

The optimization problem can be solved in a bottom-up traversal of the dendrogram. Let $\mathbf{C}_i$ be a inner node of a dendrogram, i.e. there exists a subtree, which represents the nested clusters of $\mathbf{C}_i$. Furthermore let $S_i \subset \mathcal{N}_i$ be the set of indices of the selected nested clusters. At each node, a local decision is made whether the current cluster $\mathbf{C}_i$ or the selected nested clusters $\mathbf{C}_j$, $j \in S_i$ should be selected in the flat clustering. The decision is made as follows:

$$s_i = \begin{cases} 1, & \text{if } CS(\mathbf{C}_i) > \sum_{j \in S_i} CS(\mathbf{C}_j) \\ 0, & \text{otherwise} \end{cases} \tag{6.3}$$

To be able to do a fast local decision, the total cluster stability of a subtree is propagated upwards, denoted by CS_p. Therefore, let L be the set of child nodes of the current node i. When the current cluster $\mathbf{C}_i$ is not selected, the total cluster stability of the selected nested clusters is stores in node i, otherwise the nested clusters are rejected and the cluster stability of $\mathbf{C}_i$ is kept, cf. Equation (6.4). Thus when the selection decision is made at the parent node of i, one can simply reuse the precomputed total cluster stability of the nested clusters. Figure (6.5) illustrates an example of a flat clustering.

$$CS_p(\mathbf{C}_i) = \begin{cases} \max \left\{ CS(\mathbf{C}_i), \sum\limits_{l \in L} CS_p(\mathbf{C}_l) \right\}, & \text{if } \mathbf{C}_i \text{ is an inner node} \\ CS(\mathbf{C}_i), & \text{if } \mathbf{C}_i \text{ is a leaf node} \end{cases} \tag{6.4}$$

6.3.3 The Incremental Maintenance of the Minimal Spanning Tree

The extraction of the hierarchical clustering relies on the minimal spanning tree which is built using the microclusters at a certain timestamp t. Instead of recomputing the whole minimal spanning tree, one could try to efficiently update the existing minimal spanning tree of the previous iteration. This section presents a method for computing an $\mathcal{MST}$ at time $t+1$ by updating the $\mathcal{MST}$ from timestamp t. The motivation for updating the $\mathcal{MST}$, instead of recomputing it, is obvious. The user requests for a new result are in many cases so close to each

other that many underlying, available microclusters from the previous results are still representing the distribution of the data even with very fast data streams. Usually, only a small set of new microclusters were generated and a small set of microclusters were removed. The assumption here, is that the minimal spanning tree did not change drastically within the time interval. Thus instead of recomputing the whole $\mathcal{MST}$, which is an expensive process, the presented method updates the existing $\mathcal{MST}$. In the following the method is explained in details.

Insertion and Deletion of Microclusters in $\mathcal{MRG}$

Instead of recomputing all the core-distances which are needed for building the $\mathcal{MRG}$ from scratch, it can be observed that only the core-distances of those microclusters change. This is either due to new microclusters or microclusters being deleted within the old $\text{core-dist}_{\text{minPts}}$-neighborhood [PLL07]. Figure (6.6) illustrates the two cases. An insertion of a new microcluster within the old core-distance neighborhood shrinks the core-distance of the microcluster, cf. Figure (6.6(a)). In case of a deletion of a microcluster from the old $\text{core-dist}_{\text{minPts}}$-neighborhood, the core distance grows, cf. Figure (6.6(b))

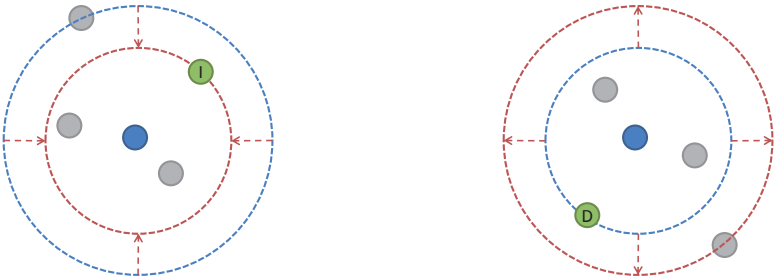
Definition 6.10 (Updating the Core-Distance: Insertion Case) *The old core-distance* $\text{core-dist}^{old}_{\text{minPts}}(mc)$ *of a microcluster* mc *needs to be recomputed only if the distance between* mc *and the newly inserted microcluster* i *is lower than* $\text{core-dist}^{old}_{\text{minPts}}(mc)$*:*

$$\text{core-dist}^{new}_{\text{minPts}}(mc) = \begin{cases} \textit{update} & \textit{if } dist_2(mc, i) < \text{core-dist}^{old}_{\text{minPts}}(mc) \\ \textit{no change} & \textit{otherwise} \end{cases} \tag{6.5}$$

In case of a recomputation of the core-distance, the new core-distance $\text{core-dist}^{new}_{\text{minPts}}(mc)$ *is smaller than* $\text{core-dist}^{old}_{\text{minPts}}(mc)$.

Definition 6.11 (Updating the Core-Distance: Deletion Case) *The old core-distance* $\text{core-dist}^{old}_{\text{minPts}}(mc)$ *of a microcluster* mc *needs to be recomputed only if the distance between* mc *and the deleted microcluster* d *is lower than* $\text{core-dist}^{old}_{\text{minPts}}(mc)$*:*

$$\text{core-dist}^{new}_{\text{minPts}}(mc) = \begin{cases} \textit{update} & \textit{if } dist_2(mc, d) < \text{core-dist}^{old}_{\text{minPts}}(mc) \\ \textit{no change} & \textit{otherwise} \end{cases} \tag{6.6}$$



(a) Insertion of a green vertex (b) Deletion of a green vertex

Figure 6.6: Illustration of the effects of inserting or deleting a microcluster, $minPts$ is set to 4. The red dashed line represents the new clustering. The insertion induces a shrinking and deletion induces a growing of the core-distance.

In case of a recomputation of the core-distance, the new core-distance $\text{core-dist}^{new}_{\text{minPts}}(mc)$ *is larger than* $\text{core-dist}^{old}_{\text{minPts}}(mc)$.

Updating the core-distances affects the mutual reachability distance between the microclusters, cf. Definition (6.2). This implies that there is no guarantee that the minimal spanning tree of the previous iteration is still valid. Thus the vertices, whose mutual reachability distance is affected, have to be considered during the update process. The updating of the minimal spanning tree consists of two phases. First, the deletion of the obsolete microclusters is performed, followed by the insertion of the newly created microclusters. In the following, the two phases are explained in more details. This order is chosen to minimize the number of vertices for the deletion case.

Deletion of Vertices From the Minimal Spanning Tree

The deletion of the obsolete microclusters leads to the deletion of the vertices which represent the microclusters in the minimal spanning tree. Removing vertices from a $\mathcal{MST}$ splits it into connected subcomponents. To connect the subcomponents again, it is not sufficient to search for the lowest weighted edges which connect the subcomponents.

Deleting a microcluster affects the core-distances of the remaining microclusters and thus directly affects the mutual reachability distance. First, the weights of the affected edges have to be updated. This is done as follows.

It can be observed that not all microclusters are affected by the deletion

of a microcluster mc_d, but only those which had mc_d in their $\text{core-dist}_{\text{minPts}}$-neighborhood are affected. This set of affected microcluster is referred to as $AFFECTED_d$. This set can be computed by a linear scan over all the microclusters. For each microcluster in $AFFECTED_d$, it is checked if the old core-distance is affected, as defined in Definition (6.11). Since only the core-distances of the microclusters in $AFFECTED_d$ are affected, only a subset of the edges from the $\mathcal{MRG}$ has to be updated. The mutual reachability distance between two microclusters that are not affected remains the same. After updating the $\mathcal{MRG}$, the microcluster mc_d is removed from the $\mathcal{MST}$. From each subcomponent, the affected microclusters $AFFECTED_d$ are removed. The reason for this is that the weights of the edges are also affected and those edges might no longer belong to the minimal spanning tree. On the other hand, the remaining connected subcomponents are still part of the minimal spanning tree since the mutual reachability distances of those microcluster have not changed. The goal is to reconnect the microclusters from $AFFECTED_d$ and the remaining connected subcomponents to form a $\mathcal{MST}$. Therefore, the approach of supervertices is used [DL01]. The reason of using this approach is to reduce the number of vertices and edges when performing Prim's algorithm in the final step. A supervertex is a vertex which represents a set of vertices which already are a connected component. In this setup, each microcluster $mc \in AFFECTED_d$ and each connected subcomponents is represented by a supervertex. Based on these supervertices, an additional complete graph is generated. The edge between two supervertices is the lowest possible weighted edge from the $\mathcal{MRG}$ which connects two vertices from the two supervertices, cf. Definition (6.12). The final step is to perform Prim's algorithm on the reduced complete graph and extract the new $\mathcal{MST}$. In the worst case, i.e. when all microclusters are affected by deleting mc_d, this approach requires the same runtime as rebuilding the $\mathcal{MST}$, since the minimal spanning tree is rebuilt over all microclusters. An example of this phase is illustrated in Figures 6.8(h)-6.8(j).

Definition 6.12 *Given a mutual reachability graph $\mathcal{MRG}(V, E)$, the edge between two supervertices sv_1 and sv_2 is defined as follows:*

$$e(sv_1, sv_2) = e(u, v), \quad \textit{with } w(e(u, v)) \leq w(e(u', v')) \tag{6.7}$$

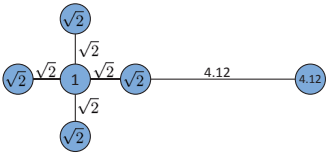
for $u, u', v, v' \in V$ and $u, u' \in sv_1$ and $v, v' \in sv_2$

Insertion of Vertices to the Minimal Spanning Tree

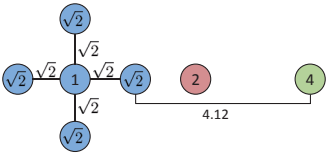
The remaining task is now to insert the newly created microclusters into the $\mathcal{MST}$ which was generated in the deletion step. Similar to the deletion case, the insertion of new microclusters affects the core-distances of the existing microclusters. The insertion case is easier to handle than the deletion case. The update of the minimal spanning tree can be realized in linear time as presented in [CH78]. The proposed algorithm in [CH78] can handle the two occurring cases in this scenario. The first case is that the insertion of a microcluster can reduce the weights of the edges in the $\mathcal{MRG}$ and thus the $\mathcal{MST}$ might no longer be correct. The second case is the insertion of the microcluster itself into the $\mathcal{MST}$. Thus the insertion of a microcluster consists of two phases. First the existing $\mathcal{MST}$ is reconstructed w.r.t. the changed mutual reachability distances due to the existence of mc_i. The second phase is responsible for the insertion of the mc_i. The complete process works a follows.

First, all the microclusters which are affected by inserting mc_i are collected and are referred to as $AFFECTED_i$. A linear scan is performed and for each microcluster, it is checked whether the old core-distance is affected, as defined in Definition (6.10). The affected microclusters have to be reinserted into the $\mathcal{MST}$. The reason is that the weight of the edges changed due to the existence of mc_i and the existing $\mathcal{MST}$ might no longer be the correct $\mathcal{MST}$ w.r.t. the new weights of the edges. The algorithm in [CH78] is able to update a $\mathcal{MST}$ when the weight of the edges decreases. A more detailed explanation of the algorithm [CH78] will be given in Section (6.4.4).

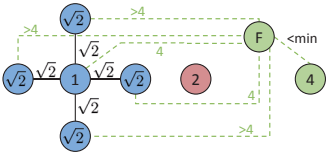
After having reconstructed the $\mathcal{MST}(V, T)$ w.r.t. the changed weight of the edges, the new microcluster mc_i is inserted into the $\mathcal{MST}$. The basic idea of [CH78] is to insert a new vertex, thus a new microcluster, to the $\mathcal{MST}$ is the following. A random vertex $r \in V$ is selected as root and at each timestamp the algorithm adds a new edge to the tree T. When a cycle is created, the largest edge from that cycle is removed. This is accomplished by a depth-first search. At the end, the result is a minimal spanning tree. The algorithm is explained in more detail in Section (6.4.4). An illustration of the insertion is represented in Figures 6.7(a)-6.8(g).



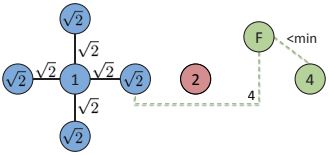
(a) Start



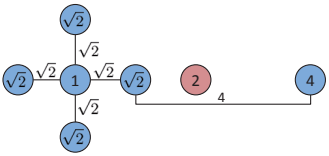
(b) New (red) vertex affects a (green) vertex



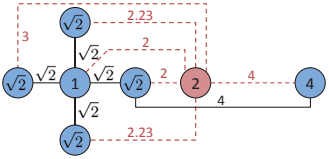
(c) Reconstruction: insert a fake vertex



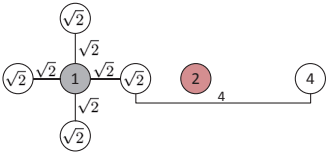
(d) Extract MST, merge the fake and the green vertex



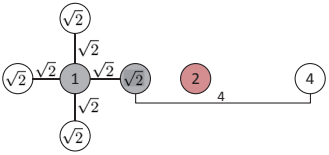
(e) MST reconstructed according to the new weight of edges



(f) New vertex's edges

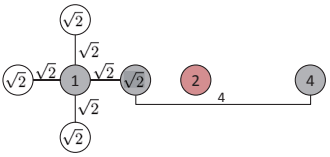


(g) Select the root node (grey)

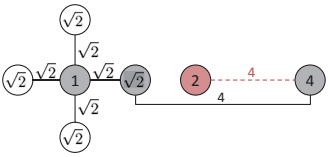


(h) Top down search

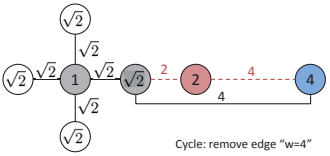
Figure 6.7: A detailed illustrates of the insertion and deletion of a vertex, where $minPts$ is set to 4. Figures 6.7(a)-6.8(g) illustrate the insertion of a vertex, which affects an existing (green) vertex and a reconstruction is invoked, followed by the insertion of the vertex. Figures 6.8(h)-6.8(j) illustrate the deletion of the same inserted vertex, where after deleting the vertex, the two supervertices are reconnected. At the end, the same $\mathcal{MST}$ as in Figure 6.7(a) is obtained.



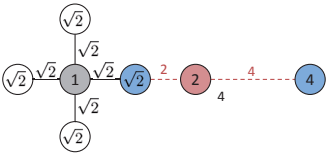
(a) Leaf node reached



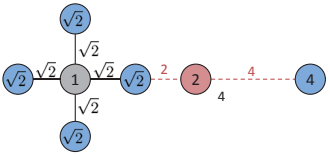
(b) Insert a new edge



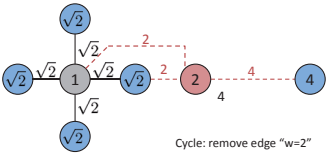
(c) Insertion of a new edge results with a cycle, remove greatest edge



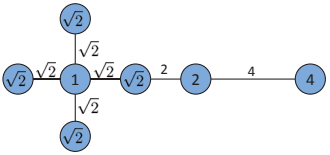
(d) Branch complete, repeat for remaining branches



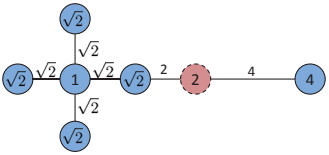
(e) All vertices processed, only root node left



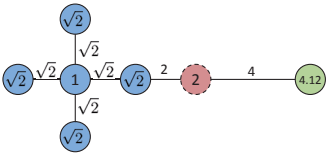
(f) Inserting a new edge, remove cycle



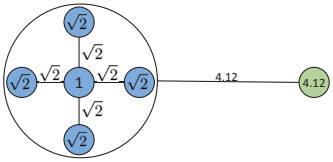
(g) Resulting MST



(h) Deletion of the (red) vertex



(i) The green vertex is affected



(j) Connect the supervertices

Figure 6.8: Continuation of the insertion/deletion steps that started in Figure 6.7.

Speed and Accuracy Tradeoff

The speed up of the incremental update comes at a cost over the correctness of the clustering. Since microclusters are representing streaming data, their centers can shift when new data is added. If the minimal spanning tree is built from scratch at timestamp t, the centers used for the distance computations are up to date w.r.t. timestamp t. The incremental variant on the other hand updates the $\mathcal{MST}$ from the previous iteration $t-1$ to a $\mathcal{MST}$ at timestamp t. Thus, the computed distances from the iteration $t-1$ might have slightly changed and the mutual reachability distances deviate from the correct values. To reduce this effect, an additional threshold can be introduced. When a the center of a microcluster has moved more than an allowed threshold within a time interval $[t-1,t]$, the microcluster has to be reinserted into the $\mathcal{MST}$. In the case of list structures the movement threshold MT can be set such that only a certain percentage of the online epsilon ε_{online} is allowed. For the index structure, it is not straight forward to set a reasonable movement threshold. One reason for that is that no reference value exists, as e.g. ε_{online} in the list structure. The tradeoff between the speed and correctness is studied in the evaluation section of the HASTREAM algorithm (Section 6.5).

6.4 The HASTREAM Algorithm

The variant which uses the index structure of the microclusters, is referred to as HASTREAM$_{IS}$, whereas the variant that uses the list structure model is referred to as HASTREAM$_{LS}$. Algorithm 6.1 illustrates the main steps of HASTREAM$_i$, with $i \in \{IS, LS\}$. HASTREAM$_i$ requires the parameter $minPts$, $minClusterWeight$ and the corresponding parameter settings for the online phase. Additionally, it uses the flag $incUpdate$ which indicates whether the algorithm should compute the offline clustering from scratch or it should maintain the minimal spanning tree from the previous iteration. The initialization step is the first phase in the algorithm. During this phase, a set of data stream objects is collected and the initial set of microclusters is generated according to the corresponding model. As long as the data stream is ongoing, the new data is processed in Steps 3-4, which represents the online phase. When a clustering is requested, the offline phase is executed, which is represented by the Steps 5-10.

Algorithm 6.1: HASTREAM$_i$(DataStream ds, $minClusterWeight$, bool $incUpdate$)

```
1: initialization phase
2: repeat
3:    get next point p ∈ ds with current timestamp t_c;
4:    process_i(p, online parameter settings);    // cf. Algorithms 6.2 and 6.4
5:    if (t_c mod updateFrequency == 0) then
6:       if incUpdate then
7:          incrementalMST_Update() //cf. Algorithm 6.8
8:       else
9:          compute MRG and corresponding MST
10:      end if
11:      HC ← extractHierarchicalClusters(MST, minClusterWeight); //cf.
         Algorithm 6.6
12:      C ← extractFlatClustering(HC);    // cf. Algorithm 6.7
13:      return C;
14:   end if
15: until data stream terminates
```

6.4.1 HASTREAM's Online Phase

In the online phase, the new stream object is processed and the microclusters are maintained. The function **process**$_i$, $i \in [LS, IS]$ in Algorithm 6.1 can be replaced by any microcluster model. Algorithm 6.2 describes the index structure model while Algorithm 6.4 describes the list model.

Algorithm 6.2: process$_{IS}$(Object p, Node $current$, bool $incUpdate$, bool $encSplit$)

```
1: decay each microcluster contained in current
2: if (current is inner node) then
3:    if current (not full or contains mc with weight w ≤ minWeight) then
4:       encSplit ← true;
5:    end if
6:    best_mc ← nearestMCwithLookAhead();    // cf. Section (7.2.1)
7:    process_LS(p, best_mc.childNode, incUpdate, encSplit);    // cf. Al-
      gorithm 6.4
8: else
9:    insert(p, current, incUpdate, encSplit);    // cf. Algorithm 6.3
10: end if
```

Algorithm 6.3: insert(Object p, Node leaf, bool $incUpdate$, bool $encSplit$)

```
1:  if current (not full or contains mc with weight w ≤ minWeight) then
2:     create new mc from p and add it to current;
3:     if incUpdate then
4:        buffer affected microclusters for MST update
5:     end if
6:  else if encSplit then
7:     create new mc from p and add it to current;
8:     if incUpdate then
9:        buffer affected microclusters for MST update
10:    end if
11:    add mc to current and call split();
12: else
13:    find nearest mc_closest and merge p into it
14: end if
```

process$_{IS}$: Index Structure Model

Algorithm 6.2 (**process**$_{IS}$) implements the index structure model to maintain the microclusters. At the start the function has, as input, the root node of the index structure and the $encSplit$ flag (cf. Chapter 7) is set to false. The top-down insertion of a new stream object can be implemented by a recursive function. First, the microclusters inside the current node are updated such that their current statistics represent the microcluster at the current timestamp.

If the current node is an inner node, cf. Steps 2-7, the next step is to check if the node still has space for a new microcluster or if any microcluster has become obsolete. If that is the case, the $encSplit$ flag is set to true. This ensures that microclusters with a lower weight than $minWeight$ are removed from the index structure at the corresponding hierarchy level l and that new microclusters can be built and inserted. Afterwards, the best fitting microcluster is identified by using the look-ahead approach, cf. Section (7.2.1). The new object is then recursively passed down to the next hierarchy level. At leaf level, cf. Step 9, the object can be processed differently.

Algorithm 6.3 (**insert**) illustrates the main steps of processing a new object at leaf level. If the current leaf node has still an empty entry or contains a microcluster whose weight is lower than $minWeight$, a new microcluster is generated from the object and is inserted into the leaf node. Depending on the chosen offline method, i.e. iterative or incremental, the newly created microcluster has to be buffered. Additionally the microclusters which are replaced at leaf level

have to be buffered as well. This allows the offline phase to update the minimal spanning tree incrementally. If the $encSplit$ flag was set on the path from the root to the leaf node, it is ensured that the current node could be split if it is overloaded. The split is then propagated up to the node where the $encSplit$ flag was initially set. A more detailed discussion about the splitting method can be found in Section (7.3.4) while introducing the LiarTree algorithm. When there is no other possibility, the object is merged into the nearest microcluster.

Algorithm 6.4: process$_{LS}$(Object p, $\varepsilon, \beta, \mu, T_p$, bool $incUpdate$)

```
1: search nearest potential microcluster mc_p;
2: merge p temporary into mc_p
3: if radius of mc_p ≤ ε then
4:    insert p into mc_p
5:    if (incUpdate and centerShift ≥ movementThreshold) then
6:       buffer mc for MST update
7:    end if
8: else
9:    search nearest outlier microcluster mc_o;
10:   merge p temporary into mc_o
11:   if radius of mc_o ≤ ε then
12:      insert p into mc_o
13:      if weight of mc_o ≥ βμ then
14:         insert mc_o into potential microcluster list
15:         remove mc_o from outlier microcluster list
16:         if incUpdate then
17:            buffer mc_p for MST update
18:         end if
19:      end if
20:   else
21:      create new outlier microcluster with p
22:   end if
23: end if
24: if (t_c mod T_p == 0) then
25:   pruning of microcluster lists, pruning();    // cf. Algorithm 6.5
26: end if
```

Process$_{LS}$: List Model

Inspired by the DenStream algorithm [CEQZ06] that introduced the model, Algorithm 6.4 (**process**$_{LS}$) illustrates the implementation of the list model to maintain

Algorithm 6.5: pruning()

```
 1: for all potential microcluster mc_p do
 2:   if weight of mc_p ≤ βμ then
 3:     remove mc_p from potential list
 4:     if incUpdate then
 5:       buffer mc_p for MST update
 6:     end if
 7:   end if
 8: end for
 9: for all outlier microcluster mc_o do
10:   ξ(t_c, t_0) = (f_2(t_c − t_0 + T_p) − 1) / (f_2(T_p) − 1)   // cf. [CEQZ06]
11:   if weight of mc_o ≤ ξ(t_c, t_0) then
12:     remove mc_o from outlier list
13:   end if
14: end for
```

the microclusters. Steps 1-7 maintain the potential microcluster list. The nearest potential microcluster is identified and the object is temporarily added to it. If the potential microclusters radius remains lower than the ε threshold , the object can be added permanently. Additionally, if the $incUpdate$ flag is set, an additional step is performed to check if the microclusters' center has moved more than a given threshold. If this is the case, the microcluster has to be buffered into a separate list, since it has to be reinserted into the maintained minimal spanning tree.

Otherwise, the nearest outlier microcluster is identified, cf. Steps 9-19, and the object is merged into it temporarily. If its radius remains lower than ε, the object is added permanently. Since a new object was added to the outlier microcluster, its weight might now be larger than $\beta\mu$ and it might become a potential microcluster. If this is the case, the microcluster is removed from the outlier list and is added to the potential list. Additionally, if the $incUpdate$ is set, the new potential microcluster needs to be buffered for a later insertion into the minimal spanning tree.

If the object did not fit in any existing microcluster, the object forms its own microcluster and is added to the outlier microcluster list. Additionally, the periodic check for pruning of the maintained microcluster list is performed, if necessary, cf. Steps 24-25.

Algorithm 6.5 (**pruning**) illustrates the pruning approach. In Steps 1-8, the potential microcluster list is scanned for microclusters, whose weights have fallen

below the threshold $\beta\mu$. Those potential microclusters become obsolete and are thus removed. If the $incUpdate$ flag is set, the microclusters are buffered to remove them from the minimal spanning tree later on. Afterwards, cf. Steps 9-13, the outlier microcluster list is scanned to find the microclusters which are most probably outliers and will not become a potential microcluster. The ξ threshold, cf. [CEQZ06] is computed for each outlier microcluster and if its current weight is lower than ξ, it is removed from the outlier list. This has no effect for maintaining the minimal spanning tree in the offline phase, thus the objects do not have to be buffered.

6.4.2 HASTREAM's Offline Phase

The offline phase of the algorithm is responsible for generating the final clustering over the set of microclusters. The offline part can be realized in two different ways. The iterative variant performs a complete computation of the clustering. The incremental variant reuses the previous computations and simply updates the existing minimal spanning tree which is used for the clustering extraction. In the following, the used function from Algorithm 6.1, Steps 5-10, is explained in more details.

Algorithm 6.6: extractHierarchicalClusters(Minimal Spanning Tree MST)

```
1:  E ← set of edges of MST
2:  all ← MST
3:  while (E ≠ ∅) do
4:    L ← collect edge(s) with the largest weight
5:    affected ← components from all containing any edge e ∈ L
6:    set scale value of current hierarchy level to the edge(s) weight
7:    subcomponents ← removing all edges e ∈ L from affected
8:    for each subcomponent c do
9:      if total weight of c < minPts then
10:       remove all edges of component c from E
11:       remove c from subcomponents
12:     end if
13:   end for
14:   components ← (components \ affected) ∪ subcomponents
15: end while
```

6.4.3 Extraction of the Hierarchical Clusters

Algorithm 6.6 illustrates the extraction of the hierarchical clusters. Given the minimal spanning tree, the algorithm starts with one connected component, representing one cluster. The largest edge(s), cf. Step 4, are collected and removed from the components which contain any of the edges. This results in connected subcomponents. Either the weight of the subcomponent is high enough w.r.t. $minPts$, and is kept, or otherwise it is rejected, cf. Steps 5-13. At the current hierarchy level of the dendrogram, cf. Step 6, the distance between the components is stored. This allows the computation of the cluster stability in later steps. This procedure is repeated until no more edges are left to process.

Algorithm 6.7: extractFlatClustering(Dendrogram den)

1: **for each** cluster $\mathbf{C}_l$ of leaf node l **do**
2: $CS_p(\mathbf{C}_l) \leftarrow CS(\mathbf{C}_l)$ and $s_l \leftarrow 1$
3: **end for**
4: **while** (current node $\neq$ root) **do**
5: **if** $CS(C_i) > \sum_{j \in S_i} CS(\mathbf{C}_j)$ **then**
6: $CS_p(C_i) \leftarrow CS_p(\mathbf{C}_i)$
7: $s_i \leftarrow 1$ and $s_j \leftarrow 0$ for all nested cluster of C_i
8: **else**
9: $CS_p(C_i) \leftarrow \sum_{j \in S_i} CS(\mathbf{C}_j)$
10: $s_i \leftarrow 0$
11: **end if**
12: **end while**
13: **return** all clusters $\mathbf{C}_r$, where $s_r = 1$

6.4.4 Extraction of the Flat Clustering

Algorithm 6.7 illustrates the extraction of the flat clustering from the dendrogram. In Steps 1-3, the cluster stability for each leaf, as presented in Definition (6.8), is initialized and each cluster is marked as selected. Afterwards in Steps 3-12, the cluster stabilities of the inner nodes are computed in a bottom-up way. For a faster computation at the current hierarchy level, the computed cluster stabilities are propagated upwards. The current cluster selection, of nested clusters, is rejected when the current stability of the cluster is higher then the selected nested clusters. In that case all nested clusters are unselected and the current

cluster is selected. Otherwise the sum of the cluster stabilities of the selected clusters is propagated upwards and the current cluster is rejected. The procedure stops at the root node.

Algorithm 6.8: incrementalMST_Update()

1: determine $AFFECTED_d$
2: $updateMST\_delete()$; // cf. Algorithm 6.9
3: determine $AFFECTED_i$
4: $updateMST\_insert()$; // cf. Algorithm 6.10
5: **return** $\mathcal{MST}$

The Incremental Update of the Minimal Spanning Tree

Instead of recomputing the minimal spanning tree $\mathcal{MST}$ each time, cf. Algorithm 6.1 Step 6, the $\mathcal{MST}$ can be incrementally updated. This is decided from the beginning using the $incUpdate$ flag. When this flag is set, the newly created microclusters and the deleted microclusters are collected during the maintenance of the microclusters, i.e. in Algorithm 6.2 or 6.2, depending the used microcluster structure. Each time a microcluster is created or deleted it is buffered in order to maintain the $\mathcal{MST}$. As described in Section (6.3.3), the procedure requires two phases. The deletion phase is illustrated in Algorithm 6.9 (**updateMST_delete**). This algorithm updates the core-distances of each affected microcluster, due to the removal of the microcluster from $AFFECTED_d$. After the removal of the microclusters, a reduced mutual reachability graph is generated, based on the supervertices composed of the remaining connected components. By applying Prim's algorithm, a new $\mathcal{MST}$ is generated on the reduced graph.

Algorithm 6.9: updateMST_delete()

1: update core distances w.r.t. $AFFECTED_d$
2: remove $AFFECTED_d$ from $\mathcal{MST}$, generate supervertices from components
3: generate complete graph $\mathcal{G}_S$ on supervertices
4: perform Prim's algorithm on $\mathcal{G}_S$
5: **return** $\mathcal{MST}$

Afterwards, the insertion phase is executed, which is illustrated in Algorithm 6.10 (**updateMST_insert**). The algorithm starts by reconstructing the $\mathcal{MST}$,

which is necessary as newly inserted microclusters can change the core-distances and thus the mutual reachability distances. Each affected vertex v from $AFFECTED_i$ is iteratively reinserted into the $\mathcal{MST}$. For each reinserted vertex v a fake vertex f is generated, which has the same edges as v to the other vertices, i.e. $w(e(v,f)) = w(e(u,f))$ for $u \in V$ and $u \neq v$. The edge $e(v,f)$ between v and f has a weight smaller than the smallest edge from the $\mathcal{MST}$. This ensures that this edge is contained in the new $\mathcal{MST}$. After performing Algorithm 6.11, the vertex f is removed, thus the edge $e(v,f)$ is also removed. The remaining task is to map the edges $e(v,f)$ and $e(u,f)$ correctly, by selecting the smaller edge w.r.t the weight.

Algorithm 6.11 illustrates the insertion of a new microcluster mc_i into the reconstructed $\mathcal{MST}$. The algorithm performs a depth first search. The algorithm needs a root vertex, which can be chosen arbitrary. The algorithm adds the new edges to the $\mathcal{MST}$. By adding edges, cycles can occur. Each time a cycle occurs, the largest edge of the cycle is removed. This procedure guarantees that the resulting set of edges does belong to the $\mathcal{MST}$ [CH78].

Algorithm 6.10: updateMST_insert()

1: update core distances w.r.t. $AFFECTED_i$
2: reconstruct $\mathcal{MST}$
3: **for each** new vertex v **do**
4: select random root vertex r
5: $insertToMST(r,v)$ // cf. Algorithm 6.11
6: **end for**
7: **return** $\mathcal{MST}$

6.5 Experimental Evaluation

Five variants of HASTREAM are compared to DenStream [CEQZ06] to test the variant that uses the list structure model of microclusters LS against the one that uses the index structure IS and the variant that uses the iterative method Ite that starts building the $\mathcal{MST}$ structure against the variants that use the incremental method Inc with different movement thresholds. Table 6.2 explains those five variants with their naming convention. All algorithms were implemented within MOA Framework [BHKP10]. Unless otherwise mentioned, the parameter settings

Algorithm 6.11: insertToMST(Vertex r, Vertex v)

```
1: mark r as processed
2: newEdge ← e(r, v)
3: for each u ∈ Adj_r do
4:   if u not processed then
5:     insertToMST(u, v)   // recursive call. To resolve cycles:
6:     l ← maximum of the edges: {t and e(u, r)}
7:     s ← minimum of the edges: {t and e(u, r)}
8:     add s to tree edges T'
9:     if w(l) < w(newEdge) then
10:      newEdge ← l
11:    end if
12:  end if
13: end for
14: t ← newEdge
```

for the parameters were set as the following. The stream speed is set to 1000 objects per time unit and the first 1000 objects are buffered for the initialization phase of the algorithms. As basis b for the decaying function $f_b(\Delta t)$, cf. Definition 5.1, b is set to 2 and the decay factor λ is set to 0.25. The window H is set to 1. For DenStream, the parameter β is set to 0.2 and the $\varepsilon_{offline}$ to perform the offline density-based clustering is set to $2 \times \varepsilon_{online}$. For HASTREAM$_{LS}$, the online parameter are set equally to those for DenStream, since both use the same structure to maintain the microclusters. For HASTREAM$_{IS}$, the maximal height parameter h is set to 4 and the minimal weight threshold ω is set to 0.5. This section is structured as follows. First the used synthetic and real world data sets are presented, followed by the evaluation measures. The last part presents the evaluation results of the different algorithm variants.

6.5.1 Datasets

2D Synthetic Dataset

The 2-dimensional dataset was generated by a modified RandomRBFGenerator provided by MOA [BHKP10]. Most of the time it consists of 4 drifting clusters with different densities. A cluster drift is initiated after each time unit. Periodically, two of the clusters move together and perform a merge. After a while the merged clusters split again and the 2 emerging clusters move back to their initial position. The duration of such a period is roughly 60 time units. The stream

Updating-Structure-Movement	Meaning
Ite-LS	iterative with list structure
Inc-LS-MT=x	incremental with list structure and movement threshold x
Inc-LS-IM	incremental with list structure and ignore movement
Ite-IS	iterative with index structure
Inc-IS-IM	iterative with index structure and ignore movement

Table 6.2: Naming convention for the different variants of the HASTREAM algorithm.

speed is set to 500 points per time unit. Furthermore the dataset contains roughly 10% noise.

10D Synthetic Dataset

The 10-dimensional dataset was generated by the implemented RandomRBFGenerator of MOA with 3 randomly drifting clusters. A cluster drift is initiated after each time unit and the stream speed was set to 500 points per time unit. Additionally each cluster has a different, randomly chosen radius. The noise in the dataset is also about 10%.

KDD CUP'99, Network Intrusion Dataset [Dat99]

Explained in previous chapters.

ICML'04, Physiological Dataset [Phy]

Also explained in previous chapters.

Forest Covertype Dataset [JAB98]

This real dataset [JAB98] contains 581012 data objects, where each data object has 10 continuous attributes with 7 labels.

6.5.2 Evaluation Measures

The clustering quality of the algorithms is evaluated using both the Cluster Mapping Measure [KKJ+11] (cf. also Section 10.2.4) and the Purity measure [ZK04]

(cf. Section 5.5.2). Additionally, to examine how strong the movement of the microclusters affects the found clustering of the incremental variant of the algorithm, the divergence of the minimal spanning tree was used as a measure. Furthermore, the runtime in seconds is used to measure the efficiency of the different algorithms.

CMM: Cluster Mapping Measure

The Cluster Mapping Measure (CMM) [KKJ+11] (cf. Section 10.2.4 for more details) considers the four following properties for evaluating the clustering of stream data.

1. **Ageing and decay:** Errors are weighted by age of corresponding object
2. **Misplaced objects:** Evolution, merging and splitting of results in overlapping of clusters may lead to misplaced objects
3. **Missed objects:** Movement of clusters may result in missing certain objects
4. **Noise:** Inclusion of noise is often inevitable

The CMM needs determines how well an object fits into a detected cluster. Additionally, the detected clusters have to be mapped to the classes of the ground truth.

Divergence of the Minimal Spanning Tree

The offline part of HASTREAM can be performed in an iterative or incremental way. As described in Section (6.4.4), the updating of the minimal spanning tree directly affects the clustering quality. Thus to examine how strong the updated minimal spanning tree deviates from the ground truth, i.e. the minimal spanning tree when computed from scratch, the following prerequisite is required. The total weight of a minimal spanning tree $\mathcal{MST}(V, T)$ is defined as

$$w_{\text{total}}(\mathcal{MST}(V,T)) = \sum_{e \in T} w(e) \tag{6.8}$$

It should be noted that in this case, the total weight is computed w.r.t. the weights of the edges and not the weights of the microclusters that have a completely different meaning. The deviation of the updated minimal spanning tree from the

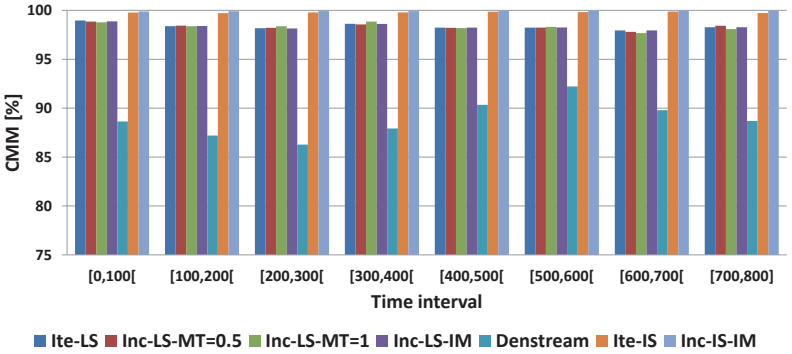


Figure 6.9: CMM: 2D Synthetic

ground truth at timestamp t is derived as follows

$$\text{divergence}_t(gt, update) = \left| 1 - \frac{w_{\text{total}}(updated)}{w_{\text{total}}(gt)} \right| \tag{6.9}$$

where gt is the ground truth $\mathcal{MST}$, computed from scratch at timestamp t, whereas $update$ is the incrementally updated $\mathcal{MST}$ at timestamp t. If the deviation is small, the clustering extracted from the updated minimal spanning tree can be expected to be more similar to the clustering extracted from the ground truth minimal spanning tree than cases with a higher deviation. Thus smaller divergence values are better.

6.5.3 Clustering Quality Results

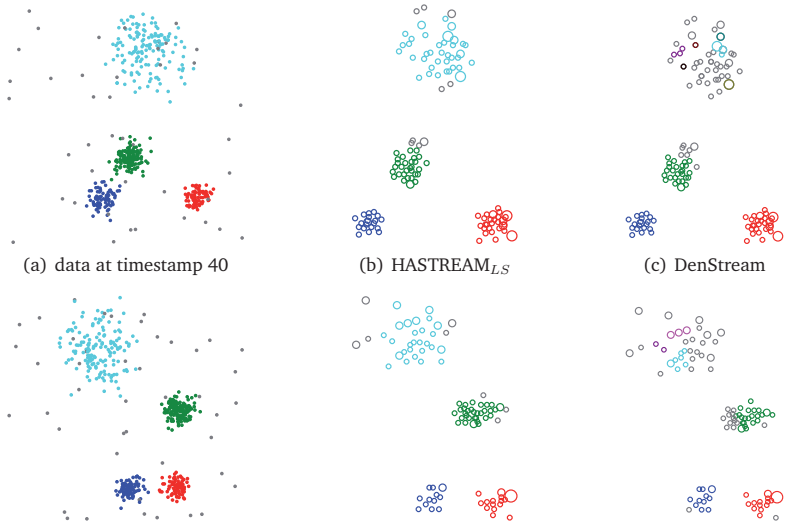
We will list these results according to the used dataset.

2D Synthetic Dataset

For this dataset the stream speed is set to 500 objects per time unit. The parameters for the online part of HASTREAM$_{LS}$ and DenStream are set to $\varepsilon_{online} = 0.015$ and $\mu = 5$. The online parameters of HASTREAM$_{IS}$ are set to $h = 3$ and $\omega = 0.5$. The parameter for the offline part of HASTREAM is set to $minPts = 5$. Additionally, the movement threshold MT for the incremental variant of HASTREAM$_{LS}$ is set to 10%, 50% and 100% w.r.t. ε_{online}.

Figure (6.9) shows the results of the averaged CMM for the two HASTREAM

variants as well as DenStream. HASTREAM outperforms DenStream at all time intervals. The reason for this is that the dataset was generated such that only one single density threshold is not able to detect all available clusters. Figure (6.10) presents ground truths, as well as clusterings at different timestamps for HASTREAM$_{LS}$ and DenStream. The results of DenStream show that the algorithm has difficulties in recognizing clusters with lower densities, due to the single density threshold. However, HASTREAM$_{LS}$ recognizes additionally less dense areas as single clusters, but this can also be a drawback if the stream is currently evolving and old microclusters still exist.



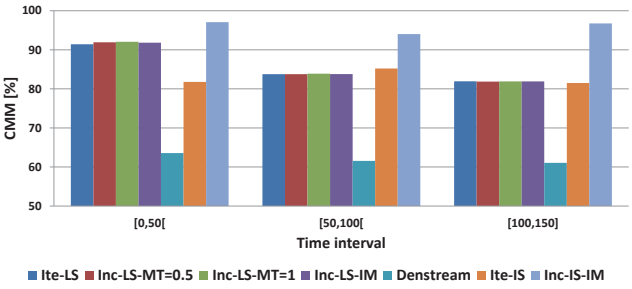
(a) data at timestamp 40 (b) HASTREAM$_{LS}$ (c) DenStream

(d) data at timestamp 160 (e) HASTREAM$_{LS}$ (f) DenStream

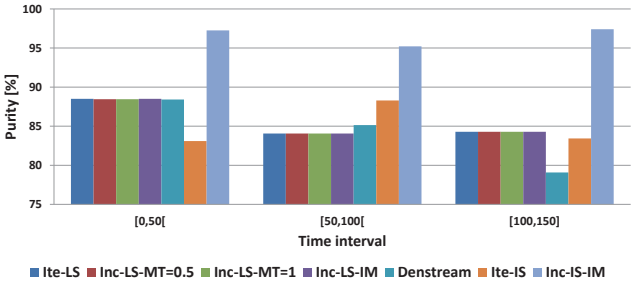
Figure 6.10: Ground truth and clustering output of both algorithm for the 2D synthetic dataset at different timestamps

10D Synthetic Dataset

The second evaluation is performed on the 10-dimensional synthetic dataset. The parameter setting of HASTREAM$_{LS}$ is $\varepsilon_{online} = 0.2$, $\mu = 5$ and $\lambda = 0.5$. The parameter setting for DenStream is set equally. The online parameters of HASTREAM$_{IS}$ are set to $h = 3$ and $\omega = 0.5$. Furthermore, the parameter for the offline part



(a)



(b)

Figure 6.11: Quality results for the 10D Synthetic Dataset: (a) CMM, (b) Purity.

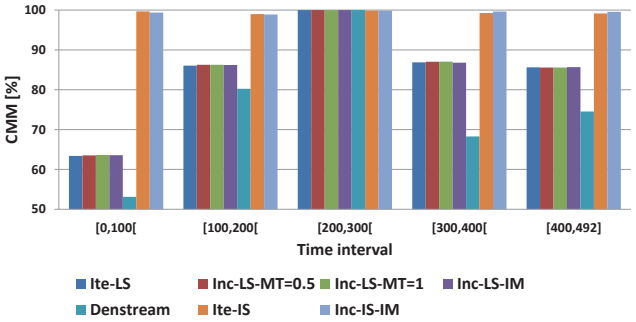
of HASTREAM is set to $minPts = 5$. The movement threshold MT for the incremental variant of HASTREAM$_{LS}$ is set once to 50% and another 100% w.r.t. ε_{online}.

Figure (6.11(a)) illustrates the averaged CMM for time intervals of length 50. It can be seen that both iterative HASTREAM variants outperform DenStream. Furthermore it should be noted that the incremental variants of HASTREAM$_{LS}$ have similar clustering quality results, whereas the incremental variant of HASTREAM$_{IS}$ deviates noticeably from its iterative counterpart.

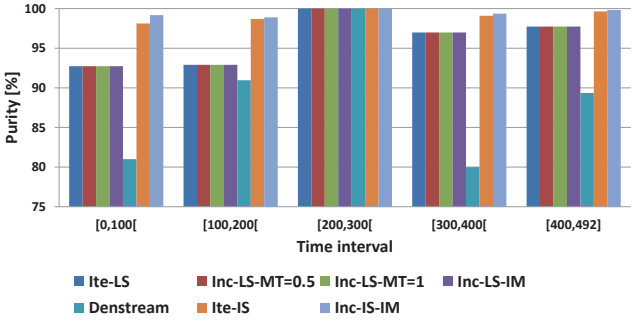
In Figure (6.11(b)), it can be seen that HASTREAM$_{LS}$ has similar clustering quality results to those of DenStream. at least one of HASTREAM variants has a considerably higher clustering purity than DenStream. Thus HASTREAM$_{LS}$ variants perform better than the iterative HASTREAM$_{IS}$ and DenStream.

Network Intrusion Dataset

For the network intrusion dataset, the parameters for DenStream are set to the same values as used in [CEQZ06]. The online settings for HASTREAM$_{LS}$ are set equally to the online parameters of DenStream. For HASTREAM$_{IS}$, the maximal height of the indexing tree is set to $h = 3$. For the offline phase of HASTREAM, the density threshold is set to $minPts = 10$. In Figure (6.12(a)), the



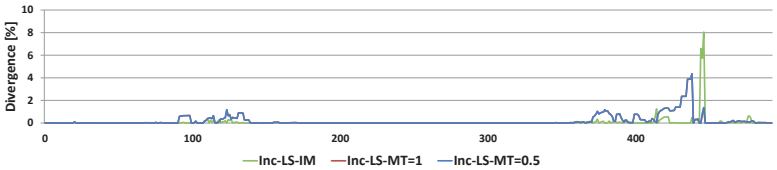
(a)



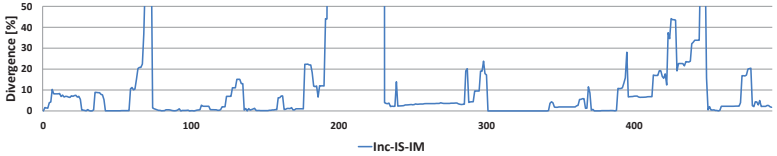
(b)

Figure 6.12: Quality results for Network Intrusion Dataset: (a) CMM, (b) Purity.

averaged CMM is shown for time intervals of length 100. It can be seen that both HASTREAM variants outperform DenStream. However, HASTREAM$_{IS}$ outperforms HASTREAM$_{LS}$ significantly. Especially in the first time interval $[0, 100]$, HASTREAM$_{IS}$ has a much higher CMM than HASTREAM$_{LS}$. This time interval contained normal connections as well as attacks of each type. One reason for



(a) Incremental MST divergence (LS): Network Intrusion Dataset



(b) Incremental MST divergence (IS): Network Intrusion Dataset

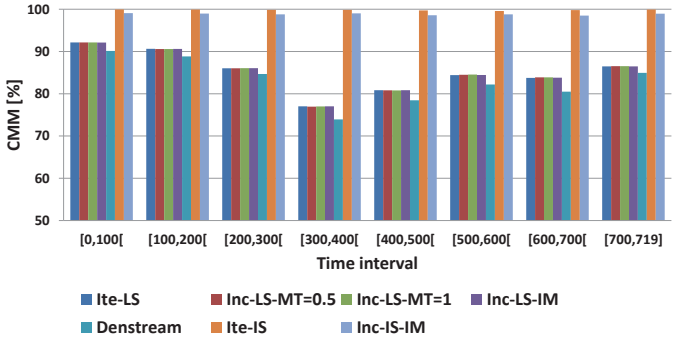
Figure 6.13: Incremental MST divergence: Network Intrusion Dataset

this high difference of CMM could be that the indexing structure is adapting faster to the evolving stream than the list structure. At time interval $[200, 300]$, all the algorithms have almost 100%. The data stream almost exclusively consists of dos-attacks, which every algorithm is able to detect. It can be seen that the results of the incremental variants are in average similar to the iterative variants of HASTREAM. The results for the purity are shown in Figure (6.12(b)). The HASTREAM variants have a higher averaged purity on each time interval than DenStream. The results of HASTREAM$_{IS}$ are again higher than the results of HASTREAM$_{LS}$. In contrast to the CMM which was low at the first time interval $[0, 100]$, the purity of the resulting clusters does not seem to be affected. Similar as for the CMM results, the purity at time interval $[200, 300]$ is almost 100%. The purity of each incremental variant is similar or equal to the iterative variant of HASTREAM. For both HASTREAM variants, the incremental variants show similar clustering quality results as their iterative counterpart. Figure (6.13) depicts how the movement of the microclusters affects the minimal spanning tree. For the network intrusion dataset, the divergence of the updated minimal spanning tree when using the list structure is low w.r.t. the ground truth, as shown in Figure (6.13(a)). The highest deviation is at the end of the data stream with around 8%. When using the index structure, cf. Figure (6.13(b)), the divergence of the updated minimal spanning tree varies significantly. At several timestamps (cf. 80 or 200), the divergence was above 100%. However this shows no strong influence on the CMM and the $purity$ measure. It should be noted that the divergence

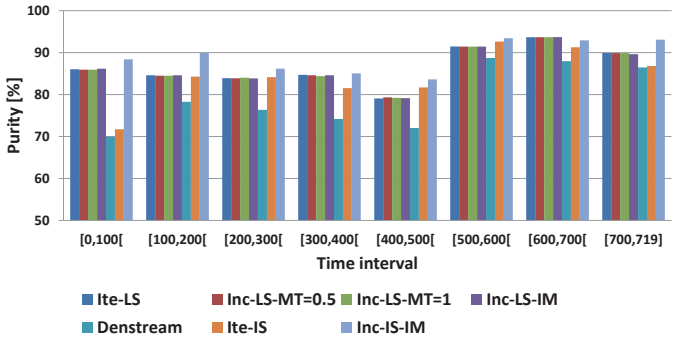
in Figure (6.13(b)) is only displayed in a range from 0% to 50% for reasons of readability. It can be seen that for this real world dataset, the clustering quality of the incremental variant is in average similar to the iterative variants of HASTREAM. However, the quality results for the HASTREAM variant using the indexing microcluster structure is higher than the variant using the list microcluster structure. This leads to the conclusion that for this dataset the index structure is more suitable to represent the data stream, than the list structure. Furthermore, this dataset shows that ignoring the movement of the microclusters when updating the minimal spanning tree can negatively affect the quality of the final clustering.

Physiological Dataset

For the next real world dataset, the online settings for HASTREAM$_{LS}$ are set to $\varepsilon_{online} = 12$, $\mu = 10$ and $\beta = 0.5$. Additionally the movement threshold MT for the incremental variant of HASTREAM$_{LS}$ is set to 50% and 100% w.r.t. ε_{online}. The online parameters for DenStream are set equally to those of HASTREAM$_{LS}$. For HASTREAM$_{IS}$: $h = 3$. For the offline phase of both HASTREAM variants, $minPts = 10$. Figure (6.14(a)) illustrates the averaged CMM for time intervals of length 100. All HASTREAM variants have higher clustering qualities than DenStream over all the stream. The averaged purity for this dataset is shown in Figure (6.14(b)). Both HASTREAM variants have a higher averaged purity than DenStream, however in contrast to the CMM results, it can be seen that these results indicate that the detected clusterings of HASTREAM$_{LS}$ have either a higher or more similar averaged purity than the clustering results of HASTREAM$_{IS}$. In this case, one can see that the movement of the microclusters can negatively affect the quality of the final clustering of the incremental HASTREAM variants, cf. incremental HASTREAM$_{IS}$. For the CMM, the averaged results for the time intervals were similar, which is not the case for the purity measure. This suggests that completely ignoring the movement of microclusters is not a viable solution and leads to wrong clusters. The purity measure of the incremental variant of the HASTREAM$_{IS}$ variant shows noticeable differences, which can be explained by the divergence of the updated minimal spanning tree w.r.t the ground truth minimal spanning tree. The divergence of the updated minimal spanning tree w.r.t. the ground truth is shown in Figure (6.15). Similar to the previous results, Figure (6.15(a)) shows that allowing a higher tolerance w.r.t. the movement of the microclusters increases the inaccuracy of the updated minimal spanning



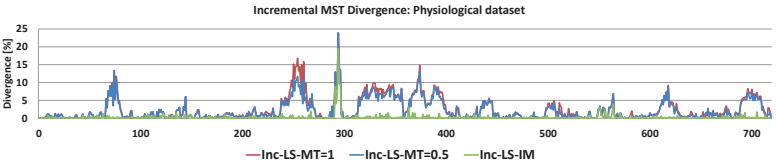
(a) CMM: Physiological Dataset



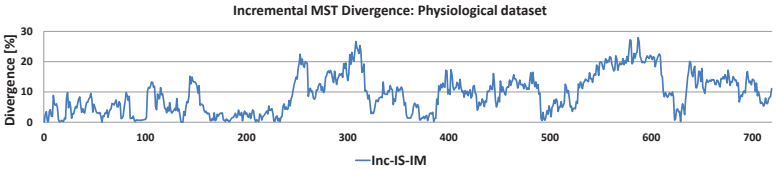
(b) Purity: Physiological Dataset

Figure 6.14: Quality results for the Physiological Dataset: (a) CMM, (b) Purity.

tree. The CMM and purity measure were not strongly affected when using the incremental HASTREAM$_{LS}$ variant. However Figure (6.15(b)) shows that the divergence of the updated minimal spanning tree of HASTREAM$_{IS}$ is high most of the time. This explains the high difference of the averaged purity, since the hierarchical clustering is extracted from the inaccurately updated minimal spanning tree. Thus, the movement can not simply be neglected when updating the minimal spanning trees of the incremental HASTREAM variants. The incremental HASTREAM$_{IS}$ variant has shown that the clustering quality of the incremental variant decreases strongly compared to the iterative variant of HASTREAM$_{IS}$. Since the movement of the microclusters can be traced more easily in the list microcluster structure, the microclusters can be reinserted and thus the divergence



(a) Incremental MST divergence (LS): Physiological dataset



(b) Incremental MST divergence (IS): Physiological dataset

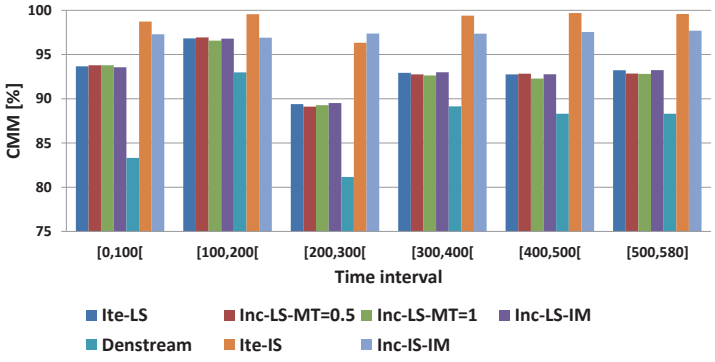
Figure 6.15: Incremental MST divergence: Physiological Dataset

of the updated minimal spanning tree is minimized w.r.t. the tolerated movement threshold. Thus the resulting extracted clustering is not strongly affected.

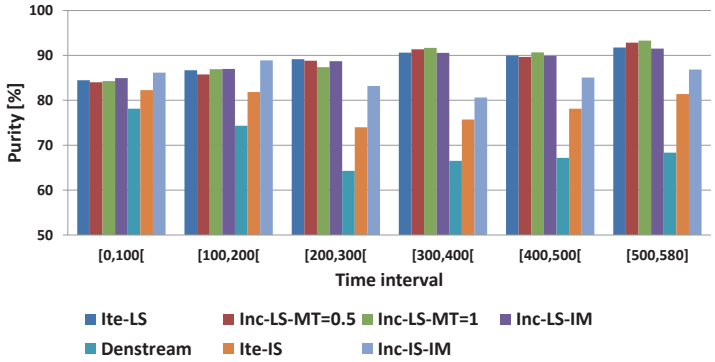
Covertype Dataset

The online parameters of HASTREAM$_{LS}$ are set to $\varepsilon_{online} = 0.2$, $\mu = 10$ and $\beta = 0.5$. The movement threshold MT of the incremental variant of HASTREAM$_{LS}$ is set to 50% and 100% w.r.t. ε_{online}. The online parameters of DenStream are set equally to HASTREAM$_{LS}$. In HASTREAM$_{IS}$, $h = 3$.The offline parameter of both HASTREAM variants is set to 10. Figure (6.16(a)) illustrates the averaged CMM results for time intervals of length 100. It can be seen that HASTREAM$_{IS}$ outperforms HASTREAM$_{LS}$ and DenStream. The averaged CMM for both HASTREAM variants is over 90% for almost each time interval.

In Figure (6.16(b)), the averaged purity is shown for the same time intervals. This results show that HASTREAM$_{LS}$ has a higher averaged purity for the resulting clusterings than HASTREAM$_{IS}$ and both have a higher averaged purity than DenStream. Similar to previous results, the results of the incremental variant of HASTREAM$_{IS}$ deviate noticeably from the ones of its iterative variant. E.g. at time interval $[200, 300]$, the averaged purity deviates roughly 10%. This can also be attributed to the divergence of the updated minimal spanning tree w.r.t. to the ground truth. In summary of the clustering quality results over all previous two synthetic datasets and three real datasets. The clustering quality of the in-



(a) CMM: Covertype Dataset



(b) Purity: Covertype Dataset

Figure 6.16: Quality results for the Covertype Dataset: (a) CMM, (b) Purity.

cremental variant of HASTREAM$_{LS}$ is similar to its iterative counterpart. The incremental variant of HASTREAM$_{IS}$ did not return reliable results due to the high divergence of the updated minimal spanning tree w.r.t. its ground truth. By comparing both iterative variants, it is not clear which microcluster data structure represented the data stream better. The indexing structure has the higher averaged CMM results, whereas the list structure has the higher averaged $purity$ results. A reason why the purity results for the list structure are higher, is that the list structure filters outlier early on, due to the differentiation between outlier and potential microclusters. The final clustering is only performed on the potential microclusters, thus reducing outliers significantly, by excluding the outlier

Algorithm	Runtime at timestamp in [ms]			
	125	250	375	492
Ite-LS	3389008	3926979	5282872	7905545
Inc-LS-MT=0.5	2666543	3147540	3909946	5486665
Inc-LS-MT=1	2663978	3143275	3931565	5530857
Inc-LS-IM	2535244	3005440	3767156	5198459
Denstream	2475282	2945663	3596089	4846652
Ite-IS	400661	494548	597135	626882
Inc-IS-IM	466571	581299	732806	768318

Table 6.3: Runtime in [ms]: Network Intrusion Dataset

Algorithm	Runtime at timestamp in [ms]			
	200	400	600	719
Ite-LS	70567	355172	514211	656283
Inc-LS-MT=0.5	70044	352738	523115	670373
Inc-LS-MT=1	75433	373991	545490	693702
Inc-LS-IM	70430	366726	537212	681969
Denstream	72209	361969	528425	672252
Ite-IS	17002	21793	27343	30522
Inc-IS-IM	17226	23052	29251	32823

Table 6.4: Runtime in [ms]: Physiological Dataset

microclusters from the final clustering, thus increasing the purity of the detected clusters.

6.5.4 Efficiency Evaluation Results

Table (6.3) illustrates the accumulated runtime of each presented algorithm. It can be seen that for this data set HASTREAM$_{IS}$ performed best and is faster than DenStream. The iterative variant is faster than the incremental variant. One reason for this could be that the cost of the search for the affected microcluster is so high that a rebuild of the minimal spanning tree is cheaper than searching the affected microclusters and updating the minimal spanning tree. Furthermore, it can be seen that the incremental variants of HASTREAM$_{LS}$ are faster than the iterative variant, but DenStream is faster than HASTREAM$_{LS}$. This data set shows that an incremental update is worthwhile, if the list microcluster structure is used.

Table (6.4) shows that the runtime is very similar for all variants using the list structure. It can be seen that the iterative variant of HASTREAM$_{LS}$ is slightly

faster than DenStream and the iterative variants are slightly slower. The HASTREAM$_{IS}$ variants have the lowest runtime and this results from the fast maintaining of the microclusters, due to logarithmic complexity of the index structure. For this data set the runtime performance for both iterative an incremental variants are very similar.

6.6 Conclusion

In this chapter, we proposed HASTREAM, a novel algorithm for hierarchical density-based clustering on evolving data streams. The presented algorithm is able to detect clusters of different densities, by adapting the density threshold for each cluster, using techniques of hierarchical clustering and graph theory. The extensive experimental evaluation study on synthetic and real world datasets shows that HASTREAM is able to find clusters of different densities, whereas the competitor DenStream fails to do so. Efficiency and effectiveness experiments shows the superiority of HASTREAM over the state-of-the-art.

Part III

Advanced Anytime Stream Clustering

Chapter 7

Outlier-Aware Non-Redundant Anytime Stream Clustering

* The varying speed of the stream is a natural characteristic of sensor data, e.g. changing the sampling rate upon detecting an event or for a certain time. In such cases, most clustering algorithms have to heavily restrict their model size such that they can handle the minimal time allowance. Recently the first anytime stream clustering algorithm has been proposed that flexibly uses all available time and dynamically adapts its model size. However, the method was not designed to precisely cluster sensor data which are usually noisy and extremely evolving. In this chapter we present the *LiarTree* algorithm that provides precise stream summaries and effectively handles noise, drift and novelty. We prove that the runtime of the LiarTree is logarithmic in the size of the maintained model opposed to a linear time complexity often observed in previous approaches.

We demonstrate in an extensive experimental evaluation using synthetic and real sensor datasets that the LiarTree outperforms competing approaches in terms of the quality of the resulting summaries and exposes only a logarithmic time complexity.

7.1 Motivation

Varying data streams (varying amount of data, varying time allowance) is a natural characteristic of sensor streaming data in many scenarios. Multiple appli-

*This chapter has been published in the Proceedings of the 5th International Workshop on Knowledge Discovery from Sensor Data (SensorKDD 2011) held in conjunction with KDD 2011 [HKS11].

cations require changing of the sampling rate of sensed data upon detecting some event or within a time period or a seasonal change. For such scenarios, budget algorithms [CKHS03] have to restrict themselves to the worst case assumption, i.e. the smallest occurring time allowance. More precisely, a budget algorithm is tailored to this minimal time allowance and will always only need (and use) this amount of time. This possibly yields large idle times. In contrast, so called anytime algorithms can provide a first result very fast and flexibly exploit additional time to improve their result. Anytime algorithms are an active field of research [KGFS10, AHWY03, YWKMN09, ADGK07, SK01, WFYH03, DeC02, SAK+09, YWKT07]. Anytime algorithms are the natural choice for varying streams, but they also outperform budget approaches on constant streams by distributing the computation time according to the confidence in the individual results [KS09, SK10]. Thus, anytime algorithms make use of any available time to deliver some result, and whenever they are given an additional time, they try to improve the quality of their results (cf. Figure 7.1).

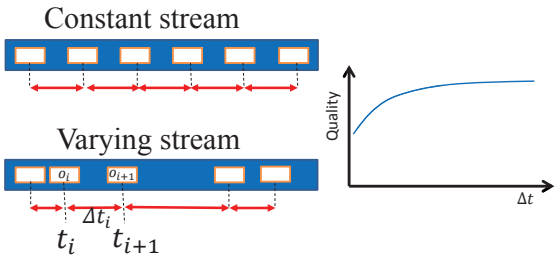


Figure 7.1: Left: for constant streams, the inter-arrival times $\Delta t_i = \Delta t_j \ \forall i, j$ while it differs with varying streams. Right: a typical quality curve of anytime mining algorithms.

In the literature, there exists a variety of anytime classification and anytime clustering algorithms over static data. However, in the streaming context, there is so far only one anytime algorithm for full-space stream clustering, which was recently proposed, called ClusTree [KABS09] (cf. Section 7.2.1). However, the algorithm does not perform any noise detection, instead, it treats each point equally. Moreover, it has limited capabilities to detect novel concepts, since new clusters can only be created within existing ones. The availability of noise and the evolving data distributions are natural characteristics of sensor data. The

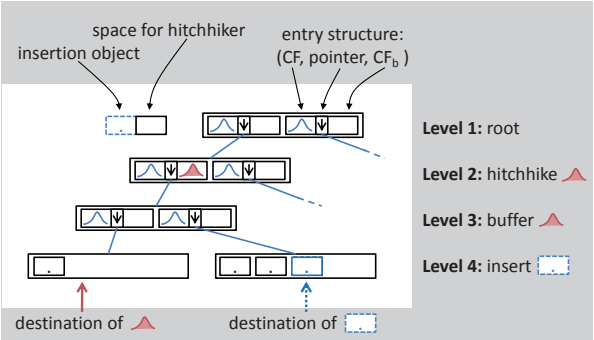


Figure 7.2: [KABS09] Illustration of the ClusTree

algorithm detailed in this chapter, the LiarTree, builds upon the ClusTree and maintains its advantages of logarithmic time complexity and self-adaptive model size. It extends its capabilities to explicitly handle noise and to better detect novel concepts.

In this chapter, we deeply detail the contained algorithms of the LiarTree, prove the logarithmic time complexity of it and perform an extensive experimental evaluation of the approach over multiple synthetic and real sensor datasets.

The chapter is organized as follows: Section 7.2 lists some of the related work by considering the ClusTree algorithm with slightly more details. Section 7.3 explains our LiarTree algorithm with all its introduced features. Section 7.4 lists the results of the extensive experimental evaluation of LiarTree, before concluding this chapter in Section 7.5.

7.2 Related work

Anytime algorithms denote approaches that are capable of delivering a result at any given point of time, and of using more time if it is available to refine the result. This is more than continuous query answering, since an anytime algorithm is arbitrarily interruptible and will still give a result. Anytime data mining algorithms such as top k processing [ADGK07], anytime learning [SK01, WFYH03] and anytime classification [DeC02, SAK+09, UXKL06, YWKT07] are an active field of research.

7.2.1 ClusTree

The ClusTree [KABS09] algorithm was the first anytime clustering algorithm for streaming data. It uses a hierarchical data structure that stores microclusters at the leaf level. The microclusters are represented through cluster features $CF = (n, LS, SS)$ that contain the number of points n, their linear sum LS and their quadratic sum SS ([ZRL96, AHWY03]). Using these cluster features the weight, mean and variance of a cluster can be computed.

Similar to other approaches, e.g. [AHWY03], the ClusTree algorithm gives more influence to recent data by weighting the objects down according to their age. It assumes that snapshots of the microclusters are taken and stored at regular time intervals t_{snap}, based on which, the influence of microclusters is determined.

Definition 7.1 ***Exponential decay.*** *Let t_{now} be the current time and t_o the arrival time of a d-dimensional object $o = (o_1, \ldots, o_d)$ with $t_o \leq t_{now}$. Then the weight of o is defined as $w(o) = 2^{-\lambda \cdot (t_{now} - t_o)}$. The time weighted cluster feature of a microcluster C is $CF_C^{(t)} = \left(n^{(t)}, LS^{(t)}, SS^{(t)}\right)$ with $n^{(t)} = \sum_{o \in C} w(o)$, $LS_i^{(t)} = \sum_{o \in C} o_i \cdot w(o)$ and $SS_i^{(t)} = \sum_{o \in C} (o_i \cdot w(o))^2$ for $i = 1 \ldots d$. A microcluster C is irrelevant if its weight $n^{(t)}$ is less than one point per snapshot t_{snap}, i.e. $n^{(t)} < 2^{-\lambda \cdot t_{snap}}$.*

The exponential decay allows the ClusTree to reuse the space taken by microclusters that became irrelevant due to their age. The structure of the ClusTree is:

Definition 7.2 ***ClusTree.*** *A ClusTree with fanout parameters m, M is a balanced multi-dimensional indexing structure with the following properties:*

- *a node contains between m and M entries*
- *an entry in an inner node of a ClusTree stores:*
  - *$CF^{(t)}$ to summarize the objects in its subtree*
  - *$CF_b^{(t)}$ to summarize the objects in its buffer*
  - *a pointer to its child node*
- *a leaf entry stores a $CF^{(t)}$ of the object(s) it represents*
- *a path from the root to any leaf node has always the same length (balanced).*

The core concept of the ClusTree is its concept of buffer and hitchhiker as illustrated in Figure 7.2. Each entry of an inner node consists of a CF representing its subtree, a pointer to the subtree and an additional buffer CF. A new object is inserted recursively into the closest entry of the current node, i.e. it follows a depth first descent. (Alternative descent strategies for the ClusTree have been discussed in [KABS09].) If the insertion of an object is interrupted before it reaches the leaf level, the object is added to the buffer of the current entry, i.e. aggregated in the buffer cluster feature $CF_b^{(t)}$ of the current entry. Hence, the space demand of a single node is constant. Hitchhiking means that an object that descends into a subtree corresponding to entry e takes e's buffer along as a hitchhiker, i.e. they descend as a tuple as long as they have the same way. To illustrate the hitchhiker concept, assume that the insertion object (drawn blue in the dashed box to the left of the root) belongs to the leaf that is marked by the dashed arrow (at the second leaf). Assume also, that the leftmost entry on the second level has a filled buffer (second distribution symbol in the entry), which belongs to a different leaf than the insertion object (indicated by the red solid arrow at the first leaf). The insertion object first descends to **level 2**, and will next descend into the left entry. It picks up the left entry's buffer in its buffer CF for hitchhikers (depicted as the solid box at the right of the insertion object). The insertion object descends to **level 3**, taking the hitchhiker along. Because the hitchhiker and the insertion object belong to different subtrees, the hitchhiker is stored in the buffer of the left entry on the **level 3** (to be taken along further down in the future) and the insertion object descends into the right entry alone to become (part of) a leaf entry at **level 4**. Once an overflow occurs on the leaf level and there is still time left, the tree grows bottom up increasing the size of the clustering model. While this allows early interrupted objects to descend further, the computational complexity is not affected, since at most two objects, i.e. the object and its hitchhiker, descend at a time. For more details please refer to [KABS09].

7.3 The LiarTree Algorithm

In this section we describe the structure and working of our novel LiarTree. In the previously presented ClusTree algorithm [KABS09], the following important issues are not addressed:

- **Overlapping:** the insertion of new objects followed a straight forward

depth first descent to the leaf level. No optimization was incorporated regarding possible overlapping of inner entries (clusters).

- **Noise:** no noise detection was employed, since every point was treated equal and eventually inserted at leaf level. As a consequence, no distinction between noise and newly emerging clusters was performed.

We describe in the following how we tackle these issues and remove the drawbacks of the ClusTree. Section 7.3.6 briefly summarizes the LiarTree algorithm and inspects its time complexity.

7.3.1 Structure and overview

The LiarTree summarizes the clusters on lower levels in the inner entries of the hierarchy to guide the insertion of newly arriving objects. As a structural difference to the ClusTree, every inner node of the LiarTree contains one additional entry which is called the noise buffer.

Definition 7.3 ***LiarTree.*** *For $m \leq k \leq M$ a LiarTree node has the structure $node = \{e_1, \ldots, e_k, CF^{(t)}_{nb}\}$, where $e_i = \{CF^{(t)}, CF^{(t)}_b\}$, $i = 1 \ldots k$ are entries as in the ClusTree and $CF^{(t)}_{nb}$ is a time weighted cluster feature that buffers noise points. The amount of available memory yields a maximal height (size) of the LiarTree.*

The noise buffer consists of a single CF which does not have a subtree underneath itself. We describe the usage of the noise buffer in Section 7.3.3.

Algorithm 7.1 illustrates the flow of the LiarTree algorithm for an object x that arrives on the stream. The variables store the current node, the hitchhiker (h) and a boolean flag indicating whether we encourage a split in the current subtree (details below). After the initialization (Lines 1 to 2), the procedure enters a loop that determines the insertion of x as follows: first the exponential decay is applied to the current node in Line 4. If nothing special happens, i.e. if none of the if-statements is true, the closest entry for x is determined (Line 8) and the object descends into the corresponding subtree (Line 24). As in the ClusTree, the buffer of the current entry is taken along as a hitchhiker (Line 23) and a hitchhiker is buffered if it has a different closest entry (Lines 9 to 12). Being an anytime algorithm, the insertion stops if no more time is available, buffering x and h in the current entry's buffer (Line 21). The issues listed in Section 7.3 are solved in the procedures $calcClosestEntry$ (Line 8), $liarProc$ (Line 6) and $noiseProc$

Algorithm 7.1: Process object (x)

```
1:  currentNode = root; encSplit = false;
2:  h = empty; // h is the hitchhiker
3:  while (true) do
4:     update time stamp for currentNode;
5:     if (currentNode is a liar) then
6:        liarProc(currentNode, x); break;
7:     end if
8:     e_x = calcClosestEntry(currentNode, x, encSplit);
9:     e_h = calcClosestEntry(currentNode, h, encSplit);
10:    if (e_x ≠ e_h) then
11:       put hitchhiker into corresponding buffer;
12:    end if
13:    if (x is marked as noise) then
14:       noiseProc(currentNode, x, encSplit); break;
15:    end if
16:    if (currentNode is a leaf node) then
17:       leafProc(currentNode, x, h, encSplit); break;
18:    end if
19:    add object and hitchhiker to e_x;
20:    if (time is up) then
21:       put x and h into e_x's buffer; break;
22:    end if
23:    add e_x's buffer to h;
24:    currentNode = e_x.child;
25: end while
```

(Line 14). We detail these methods to handle noise, novelty ($liarProc$) and drift ($leafProc$) in the Subsections 7.3.3 to 7.3.5 and describe next how we descend and reduce overlapping of clusters using the procedure $calcClosestEntry$.

7.3.2 Descent and overlap reduction

The main task in inserting an object is to determine the next subtree to descend into, i.e. finding the closest entry; Algorithm 7.2 illustrates the single steps. Besides determining the closest entry, the algorithm checks whether the object is classified as noise w.r.t. the current node and sets an $encSplit$ flag, if a split is encouraged in the corresponding subtree. The three blocks in the code correspond to the three tasks.

In the first block (Lines 1 to 6), we check whether the current node contains an irrelevant entry. This is done as in [KABS09], i.e. an entry e is irrelevant if it is

Algorithm 7.2: calcClosestEntry($node$, x, $encSplit$) // returns closest entry and marks x as noise

```
1: if (node has an irrelevant entry e_irr) then
2:    if (node is a leaf) then
3:       return (e_irr, false, false);
4:    end if
5:    encSplit = true;
6: end if
7: calculate noise probability np(x);
8: if (np(x) ≥ noiseThreshold) then
9:    mark x as noise;
10: end if
11: e_closest = closest entry;
12: if (!(node is a leaf)) then
13:    e_1 = e_closest; e_2 = 2nd closest entry;
14:    if (e_1 and e_2 overlap) then
15:       look ahead: e_i* = closest entry in e_i's child;
16:       reorganize: swap e_i* if radii decrease,
17:       update the parent cluster features of e_i;
18:       e_closest = e_1, if it contains the closest child entry; e_2 otherwise;
19:    end if
20: end if
21: return e_closest;
```

empty (unused) or if its weight $n_e^{(t)}$ does not exceed one point per snapshot (cf. Def. 7.1). In case of a leaf node, we return the irrelevant entry as the one for insertion, (Line 3), for an inner node we set the $encSplit$ flag. (Line 5).

In the second block (Lines 7 to 10), we calculate the noise probability for the insertion object and mark it as noise if the probability exceeds a given threshold. This $noiseThreshold$ constitutes a parameter of our algorithm and we evaluate its effect over our method extensively in Section 7.4.

Definition 7.4 ***Noise probability.*** *For a node* ***node*** *and an object o, the noise probability of o w.r.t.* ***node*** *is $np(o) = \min_{e_i \in \textbf{node}} \{\{dist(o, \mu_{e_i})/r_{e_i}\} \cup \{1\}\}$ where e_i are the entries of* ***node****, r_{e_i} the corresponding radius (standard deviation in case of cluster features) and $dist(o, \mu_{e_i})$ the euclidean distance from the object to the mean μ_{e_i}.*

The last block (Lines 11 to 20) finally determines the entry for further insertion. If the current node is a leaf node, we return the entry that has the smallest distance to the insertion object. For an inner node, we perform a local look

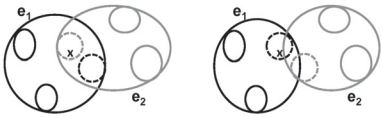


Figure 7.3: Look ahead and reorganization.

ahead to avoid overlapping, i.e. we take the second closest entry e_2 into account and check whether it overlaps with the closest entry e_1. (Line 14). Figure 7.3 illustrates an example.

If an overlap occurs, we perform a local look ahead and find the closest entries e_1* and e_2* in the child nodes of candidates e_1 and e_2 (Line 15, (dashed circles in Figure 7.3 left). Next, we calculate the radii of e_1 and e_2 if we would swap e_1* and e_2*. If they decrease, we perform the swapping and update the cluster features on the one level above (Figure 7.3 right). The closest entry that is returned is the one containing the closest child entry, i.e. e_1 in the example.

The closest entry is calculated both for the insertion object and for the hitchhiker (if any). If the two have different closest entries, the hitchhiker is stored in the buffer CF of its closest entry and the insertion object continues alone (cf. Algorithm 7.1 Line 11).

Algorithm 7.3: noiseProc ($node$, x, $encSplit$) // determines whether a noise buffer has become a cluster

```
1:  add x to node's noise buffer;
2:  if (encSplit == true) then
3:     n_avg = average weight of node's entries;
4:     ρ_avg = average density of node's entries;
5:     ρ_NB = density of node's noise buffer;
6:     if (gompertz(n_nb^(t), n_avg) · ρ_n ≥ ρ_avg) then
7:        create a new entry e_new from noise buffer;
8:        create a new empty liar root under e_new;
9:        insert e_new into node;
10:    end if
11: end if
```

7.3.3 Noise

As one output of Algorithm 7.2, we know whether the current object has been marked as noise with respect to the current node. If so, the noise procedure is called, which is listed in Algorithm 7.3. In this procedure, noise items are added to the current noise buffer and it is regularly checked whether the aggregated noise within the buffer is not noise anymore, but a novel concept. Therefore, the identified object is first added to the noise buffer of the current node. To check whether a noise buffer has become a cluster, we calculate for the current node the average of its entries' weights $n^{(t)}$, their average density and the density of the noise buffer (Lines 3 to 5).

Definition 7.5 ***Density.*** *The density $\rho_e = n_e^{(t)}/V_e$ of an entry e is calculated as the ratio between its weighted number of points $n_e^{(t)}$ and the volume V_e that it encloses. The volume for d dimensions and a radius r is calculated using the formula for d-spheres, i.e. $V_e = C_d \cdot r^d$ with $C_d = \pi^{d/2}/\Gamma(\frac{d}{2}+1)$ where Γ is the gamma function.*

Having a representative weight and density for both the entries and the noise buffer, we can compare them to decide whether a new cluster emerged. Our intuition is that a cluster that forms on the current level should be comparable to the existing ones in both aspects. Yet, a significantly higher density should also allow the formation of a new cluster, while a larger number of points that are not densely clustered are further on considered noise. To realize both criteria we multiply the density of the noise buffer with a sigmoid function, that considers the weights, before comparing it to the average density of the node's entries (cf. Line 6). As the sigmoid function we use the Gompertz function [BGH+97]

$$gompertz(n_{nb}, n_{avg}) = e^{-b(e^{-c \cdot n_{nb}})}$$

where we set the parameters b (offset) and c (slope) such that the result is close to zero ($t_0 = 10^{-4}$) if n_{nb} is 2 and close to one ($t_1 = 0.97$) if $n_{nb} = n_{avg}$ by

$$b = \frac{\ln(t_0)^{\frac{1}{1.0-(2.0/n_{avg})}}}{\ln(t_1)^{\frac{2}{n_{avg}-2}}} \qquad c = -\frac{1}{n_{avg}} \cdot \ln(-\frac{\ln(t_1)}{b})$$

Figure 7.4 depicts the variation of our Gompertz function according to n_{nb} and n_{avg}.

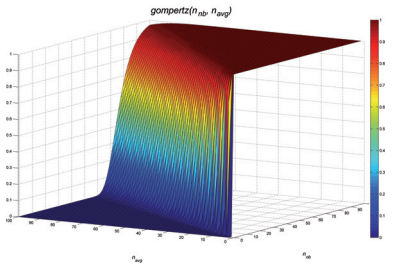


Figure 7.4: The function: $gompertz(n_{nb}, n_{avg})$.

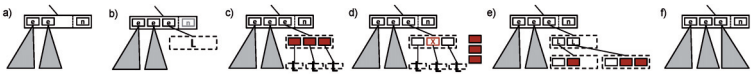


Figure 7.5: The liar concept: a noise buffer can become a new cluster and the subtree below it grows top down, step by step by one node per object.

Definition 7.6 ***Noise-to-cluster event.*** *For a node* $node = (e_1, \ldots, e_k, CF_{nb}^{(t)})$ *with average weight* $n_{avg} = \frac{1}{k} \sum n_{e_i}^{(t)}$ *and average density* $\rho_{avg} = \frac{1}{k} \sum \rho_{e_i}$ *the noise buffer* $CF_{nb}^{(t)}$ *becomes a new entry, if*

$$gompertz(n_{nb}^{(t)}, n_{avg}) \cdot \rho_n \geq \rho_{avg}$$

We check whether the noise buffer has become a cluster by now, if the encourage split flag is set to true. Note that a single inner node on the previous path with an irrelevant entry, i.e. old or empty, suffices for the encourage split flag to be true. Moreover, the exponential decay (cf. Definition 7.1) regularly yields outdated clusters. Hence, a noise buffer is likely to be checked.

If the noise buffer has been classified as a new cluster, we create a new entry from it and insert this entry into the current node. Additionally, we create a new empty node, which is flagged as *liar*, and direct the pointer of the new entry to this node (cf. Lines 7 to 9 in Algorithm 7.3). Figure 7.5 illustrates this noise to cluster event.

7.3.4 Novelty

So far, new nodes were only created at the leaf level, such that the tree grew bottom up and was always balanced. By allowing noise buffers to transform to

new clusters, we get new entries and, more importantly, new nodes within the tree. To avoid getting an increasingly unbalanced tree through noise-to-cluster events, we treat nodes and subtrees that represent novelty differently. The main idea is to let the subtrees underneath newly emerged clusters (entries) grow top down step by step with each new object, that is inserted into the subtree until their leaves are on the same height as the regular tree leaves. We call leaf nodes that belong to such a subtree *liar nodes*, the root is called *liar root*. When we end up in a liar node during descend (cf. Algorithm 7.1), we call the liar procedure which is listed in Algorithm 7.4.

Algorithm 7.4: liarProc ($liarNode$, x) // refines the model to reflect novel concepts

create three new entries with dim dimensions $e_{new}[\,]$;
for ($d = 1$ to dim) **do**
  $e_{new}[d \bmod 3].LS[d] = (e_{parent}.LS[d])/3 + \text{offset}_A[d]$;
  $e_{new}[(d+1) \bmod 3].LS[d] = (e_{parent}.LS[d])/3 + \text{offset}_B[d]$;
  $e_{new}[(d+2) \bmod 3].LS[d] = (e_{parent}.LS[d])/3 + \text{offset}_C[d]$;
  $e_{new}[d \bmod 3].SS[d] = F[d] + (3/e_{parent}.N) \cdot (e_{new}[d \bmod 3].LS[d])^2$;
  $e_{new}[(d+1) \bmod 3].SS[d] = F[d] + (3/e_{parent}.N) \cdot (e_{new}[(d+1) \bmod 3].LS[d])^2$;
  $e_{new}[(d+2) \bmod 3].SS[d] = F[d] + (3/e_{parent}.N) \cdot (e_{new}[(d+2) \bmod 3].LS[d])^2$;
end for
insert x into the closest of the new entries;
if ($liarNode$ is a liar root) **then**
  insert new entries into $liarNode$;
else
  remove e_{parent} in parent node;
  insert new entries into parent node;
  split parent node (stop split at liar root);
end if
if (non-empty liar nodes reach leaf level) **then**
  remove all liar flags in correspond. subtree ;
else
  create three new empty liar nodes under $e_{new}[\,]$;
end if

Definition 7.7 ***Liar node.*** *A liar node is a node that contains no entry. A liar root is an inner node of the liar tree that has only liar nodes as leafs in its corresponding subtree and no other liar root as ancestor.*

Figure 7.5 illustrates the liar concept, we will refer to the image when we describe the single steps. A liar node is always empty, since it has been created as an empty node underneath the entry e_{parent} that is pointing to it. Initially the liar root is created by a noise-to-cluster event (cf. Figure 7.5 b)). To let the subtree under e_{parent} grow in a top down manner, we have to create additional new entries e_i (cf. solid (red) entries in Figure 7.5). Their cluster features CF_{e_i} have to fit the CF summary of e_{parent}, i.e. their weights, linear and quadratic sums have to sum up to the same values. We create three new entries (since a fanout of three was shown to be optimal in [KABS09]) and assign each a third of the weight from e_{parent}. We displace the new means from the parent's mean by adding three different offsets to its mean (a third of its linear sum, cf. Lines 3 to 5). The offsets are calculated per dimension under the constraint that the new entries have positive variances. We set one offset to zero, i.e. $\text{offset}_A = 0$. For this special case, the remaining two offsets can be determined using the weight n_e^t and variance $\sigma_e^2[i]$ of e_{parent} per dimension as follows

$$\text{offset}_B[i] = \sqrt{\frac{1}{6} \cdot (1 - \left(\frac{1}{3}\right)^4)} \cdot (n_e^t) \cdot \sigma_e^2[i],$$

$$\text{offset}_C[i] = -\text{offset}_B[i]$$

The zero offset in the first dimension is assigned to the first new entry, in the second dimension to the second entry, and so forth using modulo counting (cf. Lines 3 to 8). If we would not do so, the resulting clusters would lay on a line, not representing the parent cluster well. The squared sums of the three new entries are calculated in Lines 6 to 8. The term $F[d]$ can be calculated per dimension as

$$F[d] = \frac{n_e^t}{3} \cdot \left(\frac{\sigma_e[d]}{3}\right)^4$$

Having three new entries that fit the CF summary of e_{parent}, we insert the object into the closest of these and add the new entries to the corresponding subtree (Lines 11 to 16). If the current node is a liar root, we simply insert the entries (cf. Figure 7.5 c)). Otherwise, we replace the old parent entry with the three new entries (cf. Figure 7.5 d)). We do so, because e_{parent} is itself also an artificially created entry. Since we have new data, i.e. new evidence, that belongs to this entry, we take this opportunity to detail the part of the data space and remove the former coarser representation. After that, overfull nodes are split (cf.

Figure 7.5 d-e)). If an overflow occurs in the liar root, we split it and create a new liar root above, containing two entries that summarize the two nodes resulting from the split (cf. Figure 7.5 e)). The new liar root is then put in the place of the old liar root, whereby the height of the subtree increased by 1 and it grew top down (cf. Figure 7.5 e)).

In the last block, we check whether the non-empty leaves of the liar subtree already reach the leaf level. In that case we remove all liar flags in the subtree, such that it becomes a regular part of the tree (cf. Line 19 and Figure 7.5 f)). If the subtree does not yet have full height, we create three new empty liar nodes (Line 21), one beneath each newly created entry (cf. Figure 7.5 c)).

7.3.5 Insertion and Drift

Once the insertion object reaches a regular leaf, it is inserted using the leaf procedure (cf. Algorithm 7.1 Line 21, detailed in Algorithm 7.5). If there is no time left, the object and its hitchhiker are inserted such that no overflow, and hence no split, occurs (Line 2). Otherwise, the hitchhiker is inserted first and, if a split is encouraged, the insertion of the hitchhiker can also yield an overflowing node. This is in contrast to the ClusTree, where a hitchhiker is merged to the closest entry to delay splits. In the LiarTree, we explicitly encourage splits to make better use of the available memory (cf. Definition 7.3). After inserting the object, we check whether an overflow occurred, split the node and propagate the split (Lines 9 to 11).

Three properties of the LiarTree help to effectively track drifting clusters. The first property is the aging, which is realized through the exponential decay of leaf and inner entries as in the ClusTree (cf. [KABS09]), (cf. Definition 7.1, a proof of invariance can be found in [KABS09]). The second property is the fine granularity of the model. Since new objects can be placed in smaller and better fitting recent clusters, older clusters are less likely to be affected through updates, which gradually decreases their weight and they eventually disappear. The third property stems from the novel liar concept, which separates points that first resemble noise and allows for transition to new clusters later on. These transitions are more frequent on levels close to the leaves, where cluster movements are captured by this process.

Algorithm 7.5: Leaf proc. ($leafNode$, x, h, $encSplit$) // inserts object x and hitchhiker h (if any) into leaf node

```
1: if (time is up) then
2:    insert x and h as entries, possibly merging closest pairs on overflow;
3: else
4:    if (node is full and encSplit == false) then
5:       merge hitchhiker to closest entry;
6:    else
7:       insert hitchhiker as entry;
8:    end if
9:    insert x as entry;
10:   if (node is overfull) then
11:      split node and propagate split;
12:   end if
13: end if
```

7.3.6 Logarithmic time complexity

We summarize the LiarTree algorithm and sketch a proof of its worst case time complexity. **Summary:** To insert a new object, the closest entry in the current node is calculated. While doing this, a local look ahead is performed to possibly improve the clustering quality by reduction of overlap through local reorganization. If an object is classified as noise, it is added to the current node's noise buffer. Noise buffers can become new clusters (entries) if they are comparable to the existing clusters on their level. Subtrees below newly emerged clusters grow top down through the liar concept until their leaves reach the regular leaf level.

Obviously the LiarTree algorithm has time complexity logarithmic in its model size, i.e. the number of entries at leaf level, since the tree is balanced (logarithmic height), the loop has only one iteration per level (cf. Alg. 7.1) and any procedure is maximally called once followed directly by a **break** statement.

Theorem 7.1 ***LiarTree time complexity*** *The clustering model $\mathcal{M}$ of a liar tree are the microclusters stored in its leaf nodes. A liar tree has by definiton a maximal height (cf. Def. 7.3) and hence its model has a maximal size $|\mathcal{M}| =: m$. The time complexity for inserting an object o into a liar tree of model size m is $O(\log m)$.*

We sketch a proof for the logarithmic time complexity of the liar tree using Algorithm 7.1.

Proof 7.1 *Let h be the height of the LiarTree, then h is logarithmic in m. The initialization takes constant time. The same holds for adding objects to cluster features (Lines 11, 19, 21 and 23) and for the noise procedure $noiseProc$ (Line 14). The two methods $liarProc$ (Line 6) and $leafProc$ (Line 17) basically have also constant complexity except for the split, which can be called maximally h times. Hence, these two methods are in $O(\log m)$. Since all three of the above methods are maximally called once per insertion object and afterwards the loop is left with a* ***break*** *statement (same lines), we are still in $O(\log m)$. We still have to poof the complexity of Lines 8 and 9 and the termination of the* ***while*** *loop. Since the look ahead is local (one level only) the $calcClosestEntry$ procedure (Lines 8 and 9) has a constant time complexity. The loop is called once per level (after each descent), i.e. it only depends on h and is therefore also in $O(\log m)$. Hence, the total time complexity of the Tree algorithm is logarithmic in the size of the clustering model, i.e. the number of maintained microclusters at the leaf level.* $\diamond$

7.4 Experimental Evaluation

To evaluate the performance of the LiarTree, we simulate different stream scenarios and compute the radii of the resulting clusters as well as the recall, precision and F1 measure. To this end, we generate synthetic data (details below), such that we know the ground truth for comparison. On synthetic data, we calculate precision and recall using a Monte Carlo approach, i.e. for the recall, we generate points inside the ground truth and check whether these are included in the found clustering, for the precision, we reverse this process, i.e. we generate points inside the found clustering and check whether they are inside the ground truth. In other words, the recall corresponds to the ground truth area that is found by the algorithm, precision corresponds to the percentage of the found area that is correct, i.e. without the unnecessary parts.

The synthetic data stream is generated using an RBF approach with additional noise, i.e. for a given number of clusters k and a given radius r we generate k hyperspheres with radius r, generate points equally at random within these spheres and add a certain percentage of noise, which is equally distributed at random in the unit cube. Novelty is simulated by adding new clusters, drift is generated by moving the cluster means along individual vectors with a given drift speed. The drift speed sets the distance that a cluster moves every 1000 points (total). If a cluster is about to drift out of the unit cube, its corresponding movement vector

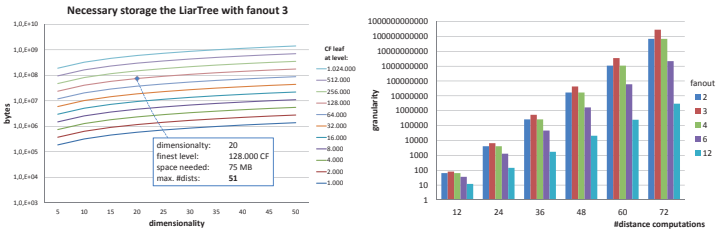


Figure 7.6: Influence of fanout and granularity.

is reflected, such that it stays inside. If not mentioned differently, we use $k = 5$, $r = 0.05$ and drift speed$= 0.02$ at 20% noise in the four dimensional unit cube. We vary the single parameters for the data stream and report the average values of the measures per algorithm.

We compare our performance to the ClusTree algorithm [KABS09], which is the only anytime stream clustering algorithm so far, and we also use the real world data employed in [KABS09]. Additionally, we test the performance using a real physiological sensor dataset [Phy]. For varying data streams, we distribute the inter arrival times of the stream objects according to a Poisson process and provide the expected arrival rate in the charts. Additionally, on constant data streams, we compare the liar tree to the CluStream approach proposed in [AHWY03] and to DenStream [CEQZ06] in the following.

We start by analyzing the influence of the fanout on the granularity and the number distance computations to reach the leaf level in Figure 7.6. Since the LiarTree extends the ClusTree, the results regarding time and space complexity are similar and can partly be transferred from the detailed analysis presented in [KABS09]. Due to the additional noise buffer, the LiarTree needs one more distance computation per node and the additional functionality, such as the liar

# MC	pps DenStream	pps CluStream	pps ClusTree	pps LiarTree
5000	2000	1500	80000	72000
2000	3700	1700	94000	84000
1000	5000	2500	105000	93000
500	7600	6500	120000	105000

Figure 7.7: Maximal points per second that can be processed by approaches for different model sizes

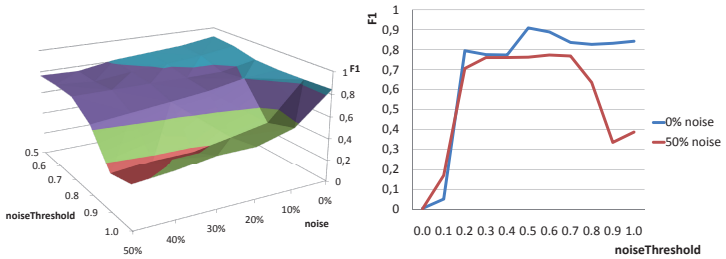


Figure 7.8: Robustness of the LiarTree to noise and the noise threshold parameter.

concept are more expensive than the simple buffering in the ClusTree. However, as shown in Section 7.3.6, the additional methods are called maximally once per object and therefore the total descend is still logarithmic. As Figure 7.6 shows, a fanout of 3 yields the highest granularity at leaf level for the liar tree. This is in accordance with the results from [KABS09], where it yielded the best trade off between space demands and computation time. Hence, we set the fanout of our LiarTree to 3, i.e. three entries (plus noise buffer) per inner node of the tree. Figure 7.7 shows for different model sizes (number of microclusters #MC) the maximal number of points per second (pps) that can be processed by the individual approaches. For CluStream and DenStream we therefore fixed the model size and counted the maximal pps, for ClusTree and LiarTree we had to fix the stream speed and measure the resulting maintainable model size.

To evaluate the noise threshold parameter of the LiarTree (cf. Section 7.3.2), the right part of Figure 7.8 shows the resulting F1 measure for 0% noise and 50% noise over the whole range of the noise threshold, the left part of the figure shows the corresponding values for all noise levels from 0% to 50% and noise thresholds from 0.5 to 1.0. The most important observation from this experiment is that the LiarTree shows good performances on a rather wide range, i.e. for a noise threshold from 0.2 up to 0.7 or 0.8. To both ends of the scale, i.e. close to zero or one, the performance drastically drops (except for 0% noise at a noise threshold close to 1.0). The performance drop for very low parameter values results from a decreasing recall, since nearly every point is considered noise in that case. For very high noise thresholds a loss in precision causes the F1 measure to drop, since new points from drifting clusters are then more likely to be added to existing microclusters rather than creating a new microcluster using the liar concept. As a consequence the area covered by the older microcluster increases

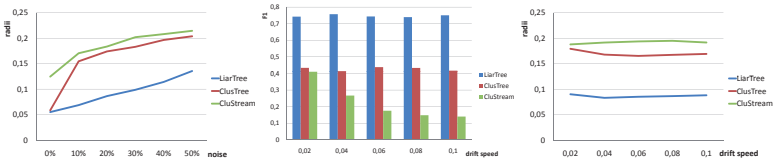


Figure 7.9: F1 measure and resulting radii for LiarTree, ClusTree and CluStream for different noise levels and drifting speeds.

and is likely to cover unnecessary parts of the data space. From the above results any choice between 0.2 and 0.8 for the noise threshold can be justified, we use 0.7 in the following. Summarizing Figure 7.8 we can notice that the LiarTree is rather robust against a reasonable choice of the noise threshold parameter.

Figure 7.9 shows the F1 and the radii of resulted clusters of LiarTree, ClusTree and CluStream for different noise values from 0% to 50%. To compare with the CluStream approach we used a maximal tree height of 7 and allowed CluStream to maintain 2000 microclusters. The parameters for the DenStream algorithm are difficult to set and greatly affect the quality of its results, such that we only used it for the performance comparison. As can be seen in the upper part of Figure 7.9, the radii of the resulting offline clusters (compare to 0.05 ground truth) of the LiarTree are considerably smaller than those of ClusTree or CluStream with the existence of noise. For 0% noise, ClusTree shows a good performance as LiarTree, while both perform considerably better than CluStream. Compact clusteres reflect less unnecessary covered area and hence improved precision. Next we evaluate the performance of the three approaches on data streams with varying drift speed. The lower part of Figure 7.9 shows the resulting values for F1 and radii. As can be seen in the lower left part, both the LiarTree and the ClusTree are not affected by higher stream speed, i.e. their F1 measure exhibits a stable value regardless of the speed. However, the LiarTree consistently outperforms the ClusTree, which proves our novel liar concept to be effective in the presence of drift and, as seen before, in the presence of noise. The main reason for the difference in the F1 measure is the poorer precision values of the ClusTree, we detail this aspect below. The CluStream approach can compete with the ClusTree for slow drift speeds in this experiment, but falls significantly behind when the drift accelerates. Its drop in performance results from both decreasing recall and precision, while the latter has clearly the stronger influence.

For the resulting radii over varying drift speeds in the bottom right part of Fig-

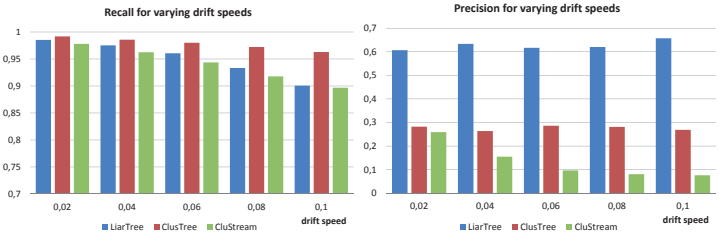


Figure 7.10: Precision and recall of the approaches for varying drift speeds.

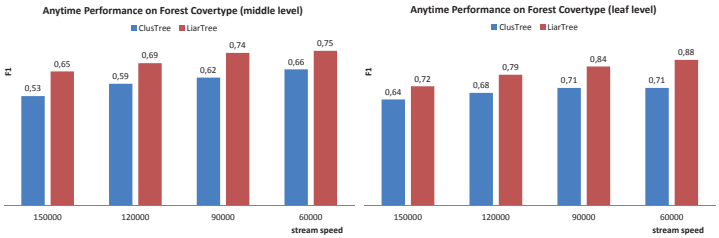


Figure 7.11: Comparing ClusTree and LiarTree on Anytime Streams using the Forest Covertype data.

ure 7.9 all three approaches show constant values over the various drift speeds, which is due to their property of removing elder data to keep track of the more important recent data. The radii resulting from the CluStream approach are two to three times larger than the ground truth. Similar values are obtained by the ClusTree for this setting, i.e. allowing a comparable number of microclusters to both approaches.

Figure 7.10 details the precision and recall values of the approaches over varying drift speeds. The left part shows that the recall values for CluStream and LiarTree slightly decrease with faster drift speeds (mind the scale compared to the right part). The reason is that both approaches adapt to the drift and delete the eldest microclusters in the process. The property of the LiarTree to actively encourage splits and create new entries can yield early outdated microclusters in some cases. In contrast, in the ClusTree, the new points are more likely to be added to existing concepts, which causes slightly increasing radii and therefore a higher recall value. However, this small benefit of the ClusTree is paid by a significantly worse precision compared to the LiarTree (cf. right part of Figure 7.10).

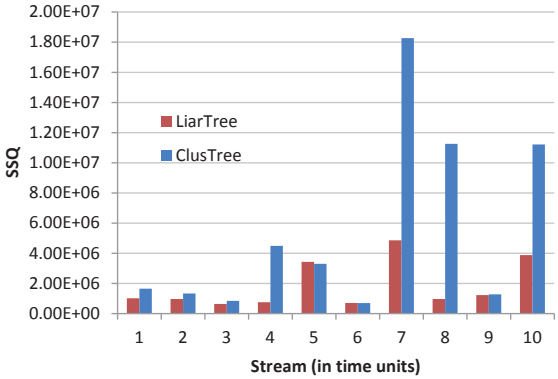


Figure 7.12: SSQ of the ClusTree vs. LiarTree using the Physiological Sensor Dataset, Stream Speed=5000

While the ClusTree can maintain its (rather low) level of precision with increasing drift speed, the CluStream approach suffers a severe loss in precision in the presence of noise and faster drifts. Once more the LiarTree clearly outperforms both approaches, showing the effectiveness of its new concepts.

Comparing LiarTree and ClusTree on varying data streams once again underlines the effectiveness of the novel concepts (cf. Figure 7.11). The employed dataset is available at [HB99] and has been used for evaluation on real data in [KABS09] and other stream clustering publications. On the x-axis the average number of points per second is reported, the arrival times were created according to Poisson distribution as mentioned above. The local noise detection and the liar concept help to better identify the underlying data distribution and yield the LiarTree to gain better results for a wide range of expected inter-arrival rates on real data.

In Figure 7.12 we evaluate the sum squared error (SSQ) [AHWY03] of LiarTree against ClusTree using the Physiological Dataset [Phy]. For this experiment, we use a constant stream speed of 5000. As sensor data like this one are naturally noisy, the LiarTree shows considerably less error than ClusTree over nearly the whole stream. This reflects the effectiveness of LiarTree when used over drifting sensor data even with a constant speed. Using the same settings as in Figure 7.12, the number of clusters detected by the LiarTree within the previous horizon was exactly the same number of classes available in it over the

whole stream. ClusTree, in contrast, detected only half the number of clusters over previous horizons in 40% of the stream. This means again that ClusTree use unnecessary spaces to cover available classes with fewer number of larger clusters, while LiarTree does not use that redundant space (consider again the results from Figure 7.9), which makes it suitable for the noisy and drifting nature of sensor data.

7.5 Conclusion

In this chapter, we detailed a novel algorithm for anytime stream clustering called *LiarTree*, which automatically adapts its model size to the stream speed in logarithmic time. It consists of a tree structure that represents detailed information in its leaf nodes and coarser summaries in its inner nodes. The LiarTree avoids overlapping through a local look ahead technique and a reorganization method. It incorporates explicit noise handling on all levels of the hierarchy. It allows the transition from local noise buffers to new entries (i.e., the microclusters) and grows novel subtrees in a top-down manner using its liar concept. This concept makes it robust against noise and changes in the distribution of the underlying stream, and thus, suitable for streaming sensor data clustering. Moreover, the LiarTree, as an anytime clustering algorithm, constitutes an anytime algorithm and automatically adapts its model size to the stream speed. In experimental evaluation, we have shown on synthetic and real sensor data for various data stream scenarios that the LiarTree outperforms competing approaches in the presence of noise and evolving data, proving its novel concepts to be effective.

Chapter 8

Anytime Subspace Stream Clustering

* A lot of research has been done in the area of full space stream clustering. To handle the varying speeds of the data stream, "anytime" algorithms are proposed but so far only in full space stream clustering. However, data streams from many application domains contain abundance of dimensions; the clusters often exist only in specific subspaces (subset of dimensions) and do not show up in the full feature space. In this chapter, the first algorithm that considers both the high dimensionality and the varying speeds of streaming data, called *SubClusTree* (**Sub**space anytime stream **Clu**stering **Tree**), is proposed. It can flexibly adapt to the different stream speeds and makes the best use of available time to provide a high quality subspace clustering. It uses a compact index structures to maintain stream summaries in the subspaces in an online fashion. It uses flexible grids to efficiently distinguish the relevant subspaces (subspaces with clusters) from irrelevant ones. In addition, it explicitly removes outliers from the clustering. The experimental results demonstrate the effectiveness and the efficiency of SubClusTree on synthetic and real datasets when compared to a state-of-the-art algorithm. They show that SubClusTree is capable of handling different stream speeds and scales well with the size of data stream and the number of dimensions. Additionally, the experiments show that SubClusTree is rigid to the introduced parameters.

*Parts of this chapter have been published in the Proceedings of the 26th Conference on Scientific and Statistical Database Management (SSDBM 2014) [HKSS14].

8.1 Motivation

There exists an ample amount of approaches for clustering over full dimensional streams and for mining clusters in subspaces of high dimensional static data. Yet only few approaches consider both perspectives; high dimensionality and streaming aspect of data. HPStream [AHWY04], HDDStream [NZP+12], PreDeConStream [HSGS12] have tried to deal with projected stream clustering. Only one approach (SiblingTree [PL07]) handles subspace clusters over *constant* data stream. Subspace clustering adds more complexity to a streaming algorithm because it has to cope with the detection of exponentially many subspace clusters. For instance, relevant subspaces have to be detected. At the same time, this task has to be done under an absence of random access, a restricted storage resources, and a limited time. Therefore, performing subspace clustering on streaming data is extremely challenging. Apparently, none of the above targeted subspace stream clustering algorithms that allow anytime clustering.

Considering all the above requirements, we propose in this chapter a first subspace anytime stream clustering algorithm called *SubClusTree*. The algorithm is based on *ClusTree* [KABS09] structure and can maintain a flexible number of the microclusters in their relevant subspaces. It, periodically, takes snapshots of microclusters to capture the evolving nature of stream and updates the relevant subspaces. Simultaneously, the subspaces, that become irrelevant, are pruned out leaving a manageable number of subspaces, each is represented by a tree. Furthermore, *SubClusTree* can explicitly detect the noise and deliver the clustering result at any point in time. The more time it has, the more dimensions it includes in the clustering. Section 5.2 lists all the related work from the area of subspace clustering, while Section 7.2 discussed the related work from the area of anytime full-space stream clustering and particularly ClusTree.

The remainder of the chapter is structured as follows: Section 8.2 introduces the basic foundations that this chapter builds on, Section 8.3 introduces our novel Algorithm *SubClusTree*, in Section 8.4, we thoroughly evaluate our approach experimentally. We conclude this chapter in Section 8.5.

8.2 Developing SubClusTree

8.2.1 Data Structure and Anytime Insertion

In our algorithm, we need a data structure that optimally manages information about different subspaces. We use a ClusTree (we denote it by *subspace tree*) to represent a single subspace. A set of dimensions needs to be associated with each subspace tree. HPStream [AHWY04] proposes *bit-vector* to keep track of the dimensionality. Therefore, we assign each subspace tree a set of dimensions in a form of a bit-vector:

Definition 8.1 (Bit-vector) *In a d-dimensional data stream, bit-vector is a d-dimensional vector which corresponds to the active dimensions (dimensions representing the corresponding subspace) in the subspace tree. Each element in this d-dimensional vector has a 1-0 value indicating whether a given dimension is active in that subspace tree.*

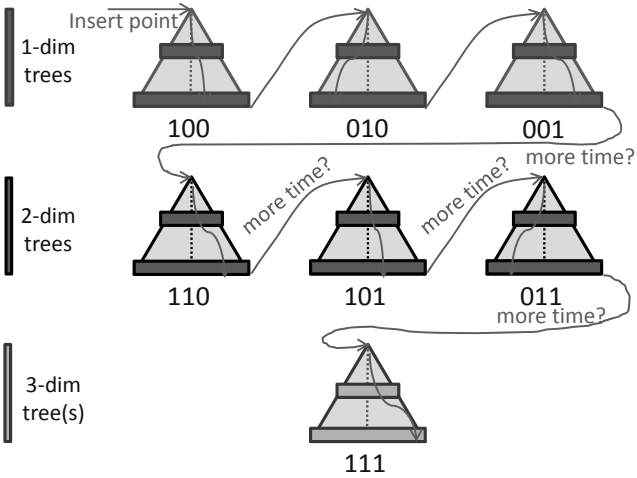


Figure 8.1: Anytime insertion and the subspace trees with bit-vector in a $3d$ space.

Figure 8.1 depicts the structure of the subspace trees. It shows an example of all subspace trees in three-dimensional space. Each tree is connected with

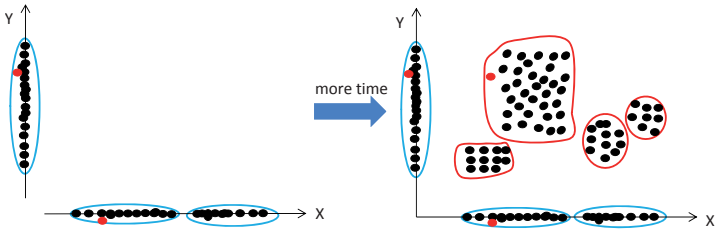


Figure 8.2: Anytime clustering behavior in a $2d$ space.

a bit-vector representing a particular subspace. Blue trees correspond to one-dimensional, black to two-dimensional, and red to three-dimensional space. Each subspace tree stores at different levels (highlighted with rectangles) the micro-clusters of different granularities. The weight of the entry is updated when we visit the corresponding node during the insertion of new point. We fixed the height of the subspace tree such that we are not interruptible within a subspace tree. In order to insert an object directly in the leaf node, our algorithm should be very quick. The ClusTree algorithm allows only a bottom-up growth of the tree structure, thus a logarithmic time complexity in the number of microclusters at the leaf level is guaranteed as the tree structure is always balanced. It was also shown that the microclusters at the lower levels still have very fine granularity. Therefore, we fixed the maximum height h_{max} of each subspace tree to 5. The point insertion till level 5 is quite fast, thus we do not interrupt the insertion within the subspace tree and always insert the point directly in the leaves. Figure 8.1 depicts the key concept of our anytime insert. We insert the point in all one-dimensional subspace trees until the leaf levels and if we have more time, we insert the point in higher dimensional trees step by step. This has a direct impact on the clustering quality. We always use the time to directly insert into the leaf level and the more time we have, the more dimensions we include in the clustering.

The clustering behavior is demonstrated in Figure 8.2. If the algorithm has less time it clusters the new point (red) in only 1D space and giving it more time results in a precise clustering in 2D space. Anytime insertion enables our algorithm to solve the challenge proposed by varying inter-arrival times of the stream.

8.2.2 Identifying Candidates for Being a Relevant Subspace Tree

For a d-dimensional space, there are 2^d subspaces, but only some of these subspaces contain the clusters. The naïve solution could be to compute all the subspaces and look for clusters in these. Though, for exponentially many subspaces, this solution is computationally not feasible. Traditional subspace clustering algorithms are of mainly two types: "bottom-up" and "top-down"[KKZ09]. The **bottom-up** algorithms start searching all one-dimensional subspaces that contain at least one cluster using a search strategy based on the A priori principle [MAK+09]. In particular, they take advantage of downward closure property of density also called *monotonicity* to reduce the search space:

Definition 8.2 (Monotonicity) *If subspace S contains a cluster, then any subspace $T \subseteq S$ must also contain a cluster. If a subspace T does not contain a cluster, then any superspace $S \supseteq T$ also cannot contain a cluster.*

In the following, we will refer to the grid-cell by "grid" for simplicity. Traditional subspace clustering algorithms like CLIQUE [AGGR98] and SCHISM [SZ05] used equal sized and fixed numbered grids to mine subspace clusters and made use of Definition 8.2. However, using these grids with microclusters has some serious drawbacks. MAFIA [NGC01] pointed out these drawbacks and suggested the usage of "adaptive" grids. It is not feasible to generate a grid for each microcluster. To group the microclusters in a meaningful way, we need an offline clustering algorithm that takes the microclusters as input and produces the final clusters. For this purpose, we choose the k-Means algorithm. To obtain better initial clustering, we pass the microcluster list to k-Means algorithm and then compute the random uniformly distributed centers from the total mean and total standard deviation of the microcluster list. This can be done as follows:

Let $CF_i^{(t)}$ denotes the cluster feature of the microcluster i, and of m microclusters. Then, using the additive property of microclusters [CEQZ06], the total features of all m microclusters can be calculated as:

$$CF_{total}^{(t)} = \left(N_{total}^{(t)}, LS_{total}^{(t)}, SS_{total}^{(t)}\right) = \sum_{i=1}^{m} CF_i^{(t)}$$

represents the total cluster feature, where $N_i^{(t)}$, $LS_i^{(t)}$ and $SS_i^{(t)}$ represent, respectively: the weighted number of the microcluster i at time t, its weighted linear

sum and its weighted squared sum. The total mean of all microclusters in the dataset μ_{total} and total standard deviation of all microclusters in the dataset σ_{total} are:

$$\mu_{total} = \frac{LS^{(t)}_{total}}{N^{(t)}_{total}} \tag{8.1}$$

$$\sigma_{total} = \sqrt{\frac{SS^{(t)}_{total}}{N^{(t)}_{total}} - \left(\frac{LS^{(t)}_{total}}{N^{(t)}_{total}}\right)^2} \tag{8.2}$$

In order to get random uniformly distributed center for each cluster, we compute the minimum and maximum range of each center. The centers should lie between $\mu_{total} - c \cdot \sigma_{total}$ and $\mu_{total} + c \cdot \sigma_{total}$, where c is a constant. It indicates that the centers lie within the c^{th} standard deviation from mean value. We have selected $c = 2$.

The next task is now to generate flexible sized grids out of these clusters. This means that we have to fit each found cluster in a grid. For this, we take the current weight, mean and standard deviation of each cluster into account. To ensure that we do not generate the empty grids, we create grids only for non empty clusters. First of all, we determine the minimum and maximum bound of the grid:

$$Min\_Bound = \mu - c \cdot \sigma \tag{8.3}$$

$$Max\_Bound = \mu + c \cdot \sigma \tag{8.4}$$

where c is as mentioned before the c^{th} standard deviation. We will keep the value of c equals to 2, because 95.6% of cluster points lie within this range and the rest can be considered as outliers which we eliminate. Please bear in mind that each grid should know its current active dimensions and its width in each dimension. The width can be calculated as follows:

$$widthPerDimension = 2 \cdot c \cdot \sigma_{dim} \tag{8.5}$$

where σ_{dim} is the standard deviation in the dimension dim. We call these grids *knownGrids*, since they are created from actual clusters. After generating the flexible grids in each single dimension ($1D$), we aim at estimating the higher subspace candidate trees (in the very first step, $2D$ candidates). A naïve solution could be to try all the combinations of *knownGrids* in different dimensions

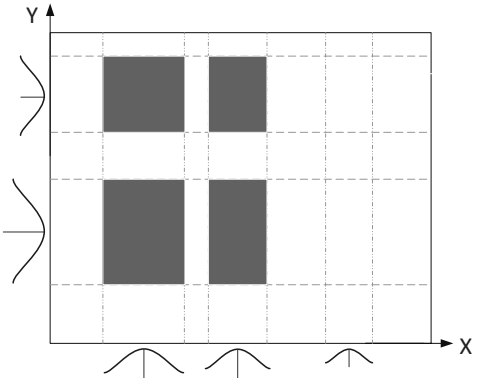


Figure 8.3: candidate grids in Subspace XY formed by combining the grids in Subspace X and Y.

and check them for clusters. Figure 8.3 shows an example of combining flexible grids in two-dimensional space. We assume that Subspace X contains 3 clusters and Subspace Y contains 2 clusters. On combining the *knownGrids* of these two subspaces, we get 4 candidate grids only in the two dimensional space, since the rightmost cluster over X does not have enough potential to be a part of a higher subspace grid.

But for a real high dimensional data, the number of combinations that we need to test is enormous. That is why we must restrict the combinations and develop a new technique that helps us to combine only meaningful grids. At this point, we simply assume that we are somehow able to compute $2D$ subspace candidates. Afterwards, we initialize these candidate trees and insert further stream points in them until a specific *batch size* is reached. The batch computation is repeated after every user defined time interval. In each computation, we first generate the *knownGrids* from so far instantiated trees and combine them to produce further candidates (in the second batch they would be $3D$ and $4D$ candidates).

8.2.3 Pruning Less Dense Candidates

To realize our ambitious concept, we have to first recognize the dense grids and subsequently solve the problem of exponential number of candidates.

Definition 8.3 (Volume) *Let dim represents a dimension of $knownGrid$ g. The volume of g is the product of width of all dimensions contained in g:*

$$V_g = \prod_{dim \in g} widthPerDimension \overset{(8.5)}{=} \prod_{dim \in g} 2 \cdot c \cdot \sigma_{dim}$$

Definition 8.4 (Density) *Let g be the KnownGrid, W_g denotes the weight of g representing the sum of the weights of all microclusters within g, and V_g be the volume of g. The density of g is defined as:*

$$dens_g = \frac{W_g}{V_g}$$

Since we combine two *knownGrids* to form a subspace candidate, we require a density estimator to estimate the density of this candidate. We also have to keep in mind that we are working in different subspaces of different dimensionalities. As a consequence, the density values are not comparable in subspace of different dimensionality [AKMS07]. Hence, we need to normalize the density estimator with their expected density. In the following, we define some notations.

Definition 8.5 (Expected density) *Let g be a knownGrid, W be the total subspace weight, and V_{sc} be the volume of SC the corresponding subspace of g (cf. Def. 8.11). The expected density of g can be defined as:*

$$expDens_g = \frac{W}{V_{sc}}$$

Definition 8.6 (Expected weight) *Let g be a knownGrid, W be the total subspace weight, V_{sc} be the volume of SC, and V_g be the volume of g. The expected weight of g is defined as:*

$$expWeight = W \cdot \frac{V_g}{V_{sc}}$$

Now we can calculate an important measure called $potential$.

Definition 8.7 (Potential) *A potential defines in general for each grid, how much the expected density of a grid is exceeded. It can be calculated by dividing the density of a grid by its expected density. In this case, we talk about knownGrid. Let g be*

knownGrid, the potential can be determined as:

$$potential = \frac{dens_g}{expDens_g} \overset{(8.4)}{=} \frac{W_g}{V_g} \cdot \frac{V_{sc}}{W} \overset{(8.6)}{=} \frac{W_g}{expWeight}$$

In order to compute the expected weight of a *knownGrid*, we need to know the total weight and the total volume of the corresponding subspace (cf. Def. 8.6). Up till now, we have no information stored to get access of total number of inserted points. For this we create a global cluster feature CF_{global} and add the newly generated stream data points to it. Additionally, we note the timestamp t of these points such that we can weigh down the global cluster feature according to its age:

$$CF^{(t)}_{global} = (2^{-\lambda t} \cdot N^{(t)} + 1, 2^{-\lambda t} \cdot LS^{(t)} + p, 2^{-\lambda t} \cdot SS^{(t)} + p^2) \tag{8.6}$$

Since subspace trees are initialized at different times, we have different total number of points inserted in these trees. The reason behind this is that we always create the *knownGrids* when a batch size is reached, afterwards, we compute subspace candidate trees and then initialize them. When we compute the expected weight of a g, we should take the initialization time of its corresponding tree and the weight of global cluster feature at that time into account and update the total weight accordingly. This is defined as follows:

Definition 8.8 *Let t_{now} be the current time and t_{init} be the initialization time of the corresponding subspace tree. We also assume that W^{init}_{global} to be the weight of CF_{global} at the initialization time, Δt denotes the time difference and W^{now}_{global} is the total weight of CF_{global} at current time, i. e.:*

$$\begin{aligned} \Delta t &= t_{now} - t_{init} \\ W^{now}_{global} &= CF_{global}.N^{(t_{now})} - W^{init}_{global} \cdot 2^{-\lambda \Delta t} \end{aligned} \tag{8.7}$$

For the computation of the total subspace volume, we again consider the CF_{global} (its standard deviation). For a given knownGrid g, we note its dimensions and

then calculate the total subspace Volume as follows:

$$V_{sc} = \prod_{dim \in g} 2 \cdot c \cdot \sigma_{dim}^{global} \tag{8.8}$$

where c is c^{th} standard deviation as mentioned previously. Now we are able to determine the expected weight and subsequently the potential of each known Grid. Figure 8.4 illustrates the behavior of potentials of g for different relevant and irrelevant dimensions from the ground truth using the N200kC3D25R13 synthetic dataset (cf. Section 8.4.1). In irrelevant dimensions, g's potential is clearly less than 1.5. In the experimental section (cf. Figure 8.10), we show this in more detail by setting different values of the potential threshold η and proving that the optimal value is equal to 1.5.

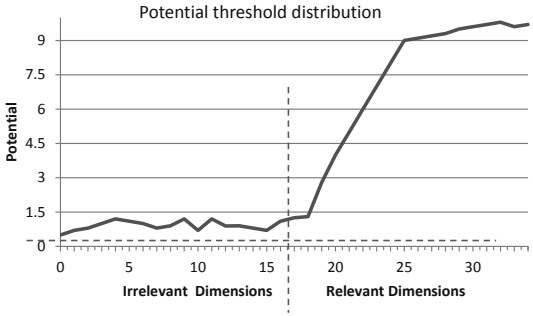


Figure 8.4: The potential of irrelevant dimensions is below 1.5, and higher for relevant ones.

Only gs with a potential threshold greater than η are generated. This implies a better storage management by pruning a lot of irrelevant gs, and also a higher quality clustering by excluding less dense or outliers gs. In the next step, we form a subspace candidate by sampling two *knownGrids* randomly. Grids with higher potential are more dense and they have really big potential to form a subspace candidate. Therefore, we influence the random selection by picking with high probability the *knownGrids* with big potential. To restrict the number of candidates, we need to estimate the expected potential of the candidate grid.

Definition 8.9 (Weight of candidate grid) *Assume that knownGrids g_1 and g_2 are forming a subspace candidate. The dimensionality of this candidate is $dim \in (g_1 \cup g_2)$. The weight of the candidate grid can be defined as follows:*

$$W_{candG} = \frac{W_{g1} \cdot W_{g2}}{W} \tag{8.9}$$

where W_{g1} and W_{g2} are weights of knownGrid g_1 and g_2 respectively. W is the weight of CF_{global} (cf. Equation (8.7)*).*

Definition 8.10 (Volume of candidate grid) *Let $candG$ be the candidate grid formed by the combination of $knownGrids$ g_1 and g_2. The volume of $candG$ is:*

$$V_{candG} = \prod_{dim \in g_1 \cup g_2} 2 \cdot c \cdot \sigma_{dim}$$

where dim is the union of the set of dimensions from g_1 and g_2, σ_{dim} represents the standard deviation in each dimension, and c is the c^{th} standard deviation.

Definition 8.11 (Volume of candidate subspace) *The dimensionality of the candidate subspace is the union of the dimensions of g_1 and g_2,and its volume is:*

$$V_{sc} = \prod_{dim \in g_1 \cup g_2} 2 \cdot c \cdot \sigma_{dim}^{global}$$

Definition 8.12 (Expected weight of candidate grid) *The expected weight of the candidate grid $candG$ can be defined in similar fashion as in Definition* (8.6)*:*

$$expWeight_{candG} = W \cdot \frac{V_{candG}}{V_{sc}}$$

where V_{sc} is the volume of the subspace $\in g_1 \cup g_2$, and W is its total weight.

Finally, the expected potential of the candidate grid can be estimated:

Definition 8.13 (Expected potential of candidate grid) *Let W_{candG} be the weight of the candidate grid $candG$ and $expWeight_{candG}$ be the expected weight of $candG$.*

$$expPotential = \left(\frac{W_{candG}}{expWeight_{candG}} \right)^{\frac{1}{n}}$$

where n represents the dimensionality of the $candG$.

Now we can sort the candidates according to their expected potential and initialize the top candidate trees. All trees that are initialized and subsequently ready to process the data stream points are called $instantiated$ trees.

8.2.4 Validating Relevant Subspace Cluster Candidates

In order to validate the instantiated trees, we first let the instantiated trees process the batch of stream points. When a batch size is reached, we generate the *macro* clusters from the leaf microclusters as explained before with k-Means algorithm. Afterwards, we create the *knownGrids* and compute their actual potential. At this point, we can assign each instantiated tree a potential, by allocating the highest potential among its *knownGrids*.

$$potential_{instTree} = \max\{potential_{g_1}, ..., potential_{g_k}\} \tag{8.10}$$

Considering this, we can rank the trees and compare their actual potential against their expected potential. We can delete an instantiated tree whose actual potential is far less than its expected potential without any concerns. Because such a tree is very unlikely to find clusters in the corresponding subspace. Furthermore, the deletion of instantiated trees helps us to better deal with the evolving nature of data stream. After some batches, we would have lots of instantiated trees and only a few of them are significant to the current clustering. Thus, it is desirable to remove non-relevant trees as the stream changes and keep only relevant trees. A relevant tree in this context is a tree with high potential. Hence, we always update the maintained trees in every batch by adding the new relevant trees and deleting the old irrelevant ones. Figure 8.5 depicts briefly the batch concept for the relevant trees in two batches. We always have to maintain the one-dimensional trees irrespective of their importance. This enables us to detect the changes in stream and to compute the higher subspace candidates from those trees.

To maintain the validity as the stream evolves, we store the subspace trees in a hash map with their bit-vector as keys. Since we have trees of different dimensionality in the same data structure, we are no more restricted to get step by step higher subspaces, instead, we can directly jump to higher relevant subspace. For example, we have trees of dimensionality 1, 2 and 3 in a hash map and later we can directly generate from these trees the candidates of dimensionality 4, 5, and 6 respectively, if there exists any.

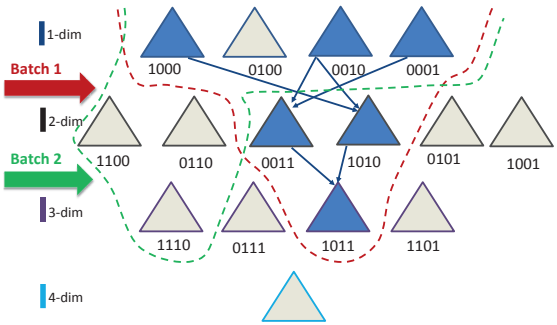


Figure 8.5: The batch processing and maintenance of relevant subspace trees for an example of two batches.

8.3 The SubClusTree Algorithm

Our algorithm describes the online component and generates the subspace trees. Any offline clustering algorithm can be applied on the final microclusters in the leaf levels of trees. While the offline part is not the idea of this work, we concentrate on the two phases of the online part.

8.3.1 Online Data Processing

Algorithm 8.1 illustrates the flow of the SubClusTree algorithm for an object x that arrives on the stream. In the first step, all one-dimensional trees are initialized and additionally a batch size is specified. The number of inserted objects is updated in Line 2. We also update the global cluster feature CF_{global} and insert the point in it. Additionally, we insert the point in all one-dimensional trees (Lines 5 to 7). These trees serve as seeds for further subspace candidates and in case of fast stream, this ensures correct clustering in all 1D spaces. In Line 8, we check whether we reach the predefined batch size. If not, then we proceed with Line 16 and insert the point in further available subspace trees as long as time allows (please note that there are no more trees before the first batch). Here we have two conditions to verify: whenever a new data object arrives, this implies that the time available for the insertion of current object is up and we have to stop and proceed with the next object. In the second condition, we check whether there are more subspace trees to process. Obviously, the insertion is done if the point is inserted in all instantiated trees. In order to handle the varying inter-arrival

Algorithm 8.1: Process Object(x) // inserts the object in all one-dimensional trees and if there is more time, the object is inserted into next higher dimensional tree

```
1: Initialization: subspaceTrees, batchSize; // subspaceTrees are stored in the
   bitmap with bit-vector as key
2: numberObjects ← numberObjects + 1 ;
3: update the global cluster feature CF_global to time t_x ;
4: W_global ← CF_global.N^{t_x};
5: for i = 1 to numberDimensions do
6:    insert x into all one-dimensional trees ;
7: end for
8: if (batchSize mod numberObjects) == 0 then
9:    knownGridMap ← createGrids(subspaceTrees, W_global) ;
10:   rank instantiated trees according to their potential ;
11:   remove trees with insufficient potential; // keep 1-D trees
12:   candidateTrees ← computeCandidates(knownGridMap) ;
13:   rank candidateTrees according to their expected potential ;
14:   initCandidateTrees(t_x, W_global, candidateTrees);
15: end if
16: while !didNewObjectArrive() and moreTreesAvailable() do
17:   insert x into next subspace tree;
18: end while
```

times of stream objects, we insert the object step by step in further subspace trees. This allows us to be flexible and to stop anywhere after inserting the point in all one-dimensional trees.

8.3.2 Batch Processing (Calculating Candidates)

By reaching the batch size, we update the candidate grids to adapt to the new distribution of the stream. However, it should be noted that the updating of the candidate grids does not need processing those (batch size) points, it is just performed after each (batch size) points over the leaf levels of all available subspaceTrees. Thus, we generate the *knownGrids* by calling the $createGrids()$ (Algorithm 8.1 Line 9). Then we rank the trees according to their potential (see Algorithm 8.2) and delete the trees with insufficient potential (Algorithm 8.1, Line 11). Afterwards, we determine subspace candidates from *knownGrids* (Algorithm 8.4) and initialize the top candidates (Algorithm 8.1, Lines 12 to 14). In $createGrids()$ (Algorithm 8.2) we create the *knownGrids* for each tree and store it in a bitmap. For the generation of *knownGrids*, we take the snapshot of microclusters stored

in the leaves of the instantiated trees and perform final clustering with k-Means (Lines 1 to 3). Then we create a *knownGrid* for each found final cluster. We store a *knownGrid* only if its computed potential (Algorithm 8.3) exceeds the predefined threshold. Otherwise, this grid is not important for the clustering and is considered as outlier. For each tree, we create a list of *knownGrids* and store it in the bitmap (Lines 1 to 11).

Algorithm 8.2: createGrids($subspaceTrees, W_{global}$) // generates grids from final clusters

```
1:  for each tree : subspaceTrees do
2:    mcList ← get leaf microclusters from tree ;
3:    clusterList ← kMeans(mcList);
4:    for each cluster : clusterList do
5:      create a grid knownGrid;
6:      potential ← computePotential(knownGrid, W_global);
7:      if potential ≥ η then
8:        knownGridList ← add knownGrid to the list ;
9:      end if
10:   end for each
11:   knownGridMap ← put knownGridList in the corresp. dim. ;
12: end for each
13: return knownGridMap
```

The potential of a knownGrid is computed in the *computePotential()* (Algorithm 8.3). For this, we consider the weight, volume and dimensionality of the grid and compute the expected weight of the grid using Definition (8.6). Then we calculate the potential of this grid using Definition (8.13).

The subspace candidates are computed in *computeCandidates()* (Algorithm 8.4). We first sample two *knownGrids* g_1 and g_2 randomly but with high probability dense grids, i.e. grids with high potential. In order to form a candidate, these two grids must have different dimensions. The sampling (Algorithm 8.4, Line 2) is performed globally randomly but with high probability for dense grids. However, higher dimensional candidates are formed only one time after checking the bit-vector. This means, once for instance a $4D$ candidate grid say (110101) is formed by combining, say, the $2D$ grid: (110000) and the $2D$ grid: (000101), it will be directly excluded and will not be generated again when say combining the $3D$ grid: (110100) with the $1D$ grid: (000001). We then calculate the weight, expected weight of the candidate grid as defined in Definitions (8.9) and (8.12).

Finally, we can compute the expected potential of the candidate as explained in Definition (8.13) (cf. Line 11). To overcome the issue of exponential many candidates, the candidates are ranked according to their expected potential.

Algorithm 8.3: computePotential($knownGrid, W_{global}$) // computes the potential of a *knownGrid*

1: $g \leftarrow knownGrid$;
2: $W_g \leftarrow g.getWeight()$;
3: $V_g \leftarrow g.getVolume()$ /* cf. Definition (8.3) */;
4: $dim_g \leftarrow g.getDimensions()$;
5: $n \leftarrow g.getDimensions.length$;
6: $V_{SC} \leftarrow getSpaceVolume(dim_g)$;
7: $expWeight_g \leftarrow W_{global} \cdot \frac{V_g}{V_{SC}}$ /* cf. Definition (8.6) */;
8: $potential = \left(\frac{W_g}{expWeight_g}\right)^{\frac{1}{n}}$ /* cf. Definition (8.13) */ ;
9: **return** $potential$

Algorithm 8.4: computeCandidates($knownGridMap$)// samples *knownGrids* and returns subspace candidates with expected potential

1: **while** there are non-generated possible candidates **do**
2: sample two *knownGrids* g_1 and g_2 randomly with different dimensions;
3: $dim_{candG} \leftarrow g_1.getDimensions()$ **bitwise or** $g_2.getDimensions()$;
// subspaceTrees are stored in the bitmap with bit-vector as key
4: $n \leftarrow dim_{candG}.length()$;
5: $W_{g_1} \leftarrow g_1.getWeight()$;
6: $W_{g_2} \leftarrow g_2.getWeight()$;
7: $W_{candG} \leftarrow \frac{W_{g_1} \cdot W_{g_2}}{W_{global}}$ /* cf. Definition (8.9) */;
8: $V_{SC} \leftarrow getSpaceVolume(dim_{candG})$ /* cf. Definition (8.11) */;
9: $V_{candG} \leftarrow \prod getWidthPerDimension()$ /* cf. Definition (8.10) */;
10: $expWeight_{candG} \leftarrow W_{global} \cdot \frac{V_{candG}}{V_{SC}}$ /* cf. Definition (8.12) */;
11: $expPotential \leftarrow \left(\frac{W_{candG}}{expWeight_{candG}}\right)^{\frac{1}{n}}$ /* cf. Definition (8.13) */;
12: **end while**
13: **return** list of dim_{candG} with their $expPotential$

8.4 Experimental Evaluation

In this section, we present the experimental evaluation of SubClusTree. We tested the algorithm on synthetic and real datasets. To check the anytime capability of our approach realistically, we modeled the data stream with varying inter-arrival times of objects.

8.4.1 Experimental Setup

We implemented the SubClusTree algorithm in java 1.7. All the experiments were done on a 64 bit Windows 7 machine with Intel Core i5 2.50 GHz processors and 4 GB RAM. Similar to the method in Section 7.4, we have used the Poisson process to vary the inter-arrival times of objects in the stream. The expected inter-arrival time of an exponentially distributed random variable with parameter λ_p is: $E[t] = \frac{1}{\lambda_p}$. In the experiments, we vary the value of λ_p to achieve different average speeds.

For the experiments, we generated a synthetic dataset using an RBF (Radial basis function) generator. We also added 5% noise randomly to our data generator. We use the following notation: N represents the total number of data objects, D indicates the total dimensionality of dataset, C stands for total number of natural clusters, and R indicates the number of relevant dimensions. Thus, N200kC3D25R13 for instance stands for 200,000 data objects, belonging to 3 different clusters with 13 relevant dimensions out of total 25 dimensions. Two real datasets were used for testing the different properties of SubClusTree: the Network Intrusion Dataset [Dat99] and the Covertype Dataset [JAB98]. Most of the stream clustering algorithms use "Purity" [ZK04] as a very common and intuitive measure. Purity captures the majority class in each cluster and calculates its dominance. For a set K of clusters, the purity of the clustering is computed as the weighted average purity of all clusters in K:

$$purity = \sum_{k=1}^{K} \frac{n_k}{n} \cdot \frac{max_c(n_{ck})}{n_k} = \frac{1}{n} \sum_{k=1}^{K} max_c(n_{ck})$$

where n_k is the number of objects in the cluster k, n_{ck} gives the number of objects with class label c in cluster k, and n is the total number of objects $n = \sum_{k \in K} n_k$.

In the experiments, we fixed the value of $\eta = 1.5$, we show the effect of η on clustering quality by varying its value and prove that 1.5 yields the best result.

Batch size specifies how frequent SubClusTree updates its subspace candidates. In the experiments, we fixed the batch size to 10,000. We justify this setting by showing its impact on clustering quality. Unless particularly mentioned, the stream parameters were set as follows: *window size* $w = 10,000$ (related to the experiment over the stream, represents how many previous points will be considered from the stream that end with the current timestamp), decay rate $\lambda = 1$, *horizon* $H = 1$. For the evaluation of clustering quality, we applied k-Means algorithm on leaf clusters of each tree with $k = 10$. The maximum height of the subspace tree h_{max} was fixed to 5.

8.4.2 Results

In this section we perform the experiments. In the experiments we address the following issues:

1. Evaluation of the clustering quality on both synthetic and real datasets, and comparing SubClusTree performance with a competing stream projected clustering algorithm PreDeConStream [HSGS12](cf. Section 8.4.2)

2. Sensitivity analysis: the selection of optimal potential threshold, batch size and window size; how does they affect the clustering quality? (cf. Section 8.4.2)

3. Scalability test: the scalability in terms of number of dimensions D, number of relevant dimensions R, and number of clusters C

4. The ability to detect irrelevant dimensions as the ground truth of synthetic data changes

5. Anytime capability: the effect of stream speed on clustering quality and the efficiency of the algorithm with fast streams.

Evaluation of Clustering Quality

Figure 8.6 shows the final clustering purity at different timestamps using the N200KC3D25R13 synthetic dataset . At the beginning, only one-dimensional trees are initialized and after the insertion of 10,000 points, we tested the purity of these trees. All the trees in relevant dimensions have average weighted purity of 55% to 60%. At timestamp 20,000, the candidate two-dimensional trees are

initialized and consequently their purity at that time is not very high. Trees of different dimensionality are always updated after every 10,000 objects. Therefore, the purity increases slowly but becomes stable to 90% at timestamp 70,000. The purity of trees representing the ground truth clusters were above 95%, but to be fair, we averaged the purity of all subset of relevant trees. That is why the maximal purity is 0.92$\%$. In the next experiment, we tested the clustering quality

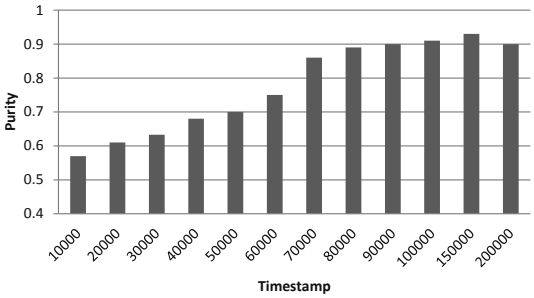


Figure 8.6: Clustering quality (Synthetic dataset N200KC3D25R13, $H = 1$, $w = 10,000$).

on Network Intrusion Dataset. The window size w was chosen to be 1000 and horizon H was 10. Figure 8.7 demonstrates the average purity at different points of time. As we can see, there are many moments where the average purity is almost 100%. This is because all instances of the dataset at those times belong to the same connection type (normal). At timestamp 150K, a total of 12 attacks were recorded. This represents a big concept drift, and a direct consequence of it is that many subspaces that were previously relevant become irrelevant and vice versa. The trees, that were running previously, had a clear fall in their purity but still the average purity of microclusters was above 70$\%$. Another peak time of attacks was timestamp 350K, and 450K. We also tested the clustering quality of SubClusTree for Forest Covertype dataset. The window size was 10,000 and horizon was 1. Figure 8.8 depicts the cluster purity during the whole period of stream execution. On reviewing the purity of every single instantiated tree, we found that some subspace trees had a purity above 90$\%$ and at the same time the purity of other trees were below 60%. We assume that those trees represented the irrelevant subspaces. Since we do not have ground truths to confirm this assumption, we fairly averaged the purity of all trees.

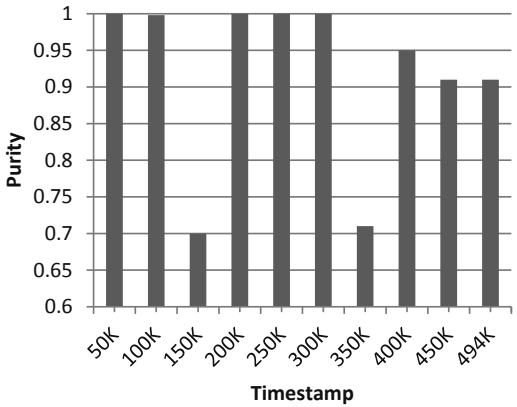


Figure 8.7: Clustering quality (Network Intrusion dataset, $H = 10$, $w = 1000$).

To evaluate the performance with a state-of-the-art algorithm, we compared SubClusTree with PreDeConStream [HSGS12] (cf. Chapter 5). As mentioned in Chapter 5, PreDeConStream is a projected stream clustering algorithm. Due to the fact that, to the best of our knowledge, no available implementable subspace stream clustering algorithm is available yet, we have compared our algorithm against this recent projected clustering algorithm. As a projected clustering algorithm, PreDeConStream tries to find the biggest possible cluster by expanding it over all subspaces, without any redundant representation of any object. Subspace clustering algorithms on the other hand, as SubClusTree, try to find all clusters within all relevant subspaces without connecting them, for a better description of all clusters available within different subspaces. However, this yields a redundancy in representing points in many subspace clusters. Thus, by default, the *total* number of found clusters in SubClusTree is considerably bigger than that of PreDeConStream. And a straightforward comparison between them will be unfair against our subspace clustering algorithm. Therefore, for a fair comparison, we have compared the purity of PreDeConStream with three types of purities of clusterings delivered by SubClusTree:

- **maximum:** representing the purity of the clustering of the *most pure* delivered subspace
- **average:** representing the average of all purities of all found clusterings in

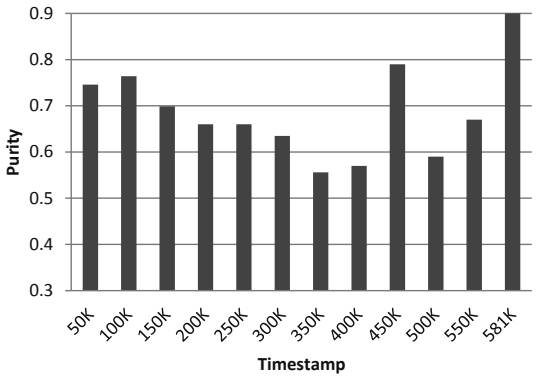


Figure 8.8: Clustering quality (Forest Cover dataset, $H = 1$, $w = 10,000$).

delivered subspaces (similar to these experiments in Figures 8.7 and 8.8)

- **highest subspace:** representing the purity of the delivered subspace with the highest dimensionality

We have set the parameters as $H = 10$ and $w = 1000$ for both PreDeConStream and SubClusTree. For PreDeConStream, we set $\varepsilon_N = 16$, $\beta = 0.2$, $\mu_F = 10$, $\lambda = 0.01$, $Init = 1000$, $\tau = 32$ and $\kappa = 4$. Figure 8.9 depicts the clustering purity of the above three clusterings delivered by SubClusTree compared with that of PreDeConStream at different timestamps of the Network Intrusion dataset. Table 8.1 lists, at the specific timestamps, the subspaces which contain the clusterings delivered by SubClusTree where the purities: *maximum* and *highest dimensionality* where calculated. It can be seen from Figure 8.9 that in all of the selected timestamps, the maximum purity is higher than that of PreDeConStream. It can be also seen that the average and the highest dimensional purities are in most of the time of higher purity than PreDeConStream. This reflects the fact that SubClusTree finds more pure clusters over all subspaces even when compared with a projected clustering algorithm, and that the purity of found clusters is high even when found at high dimensional subspaces.

Sensitivity Analysis

Figure 8.10 depicts the resulting average purity for various values of η. It shows that the SubClusTree is stable when we vary this parameter. If we set the value

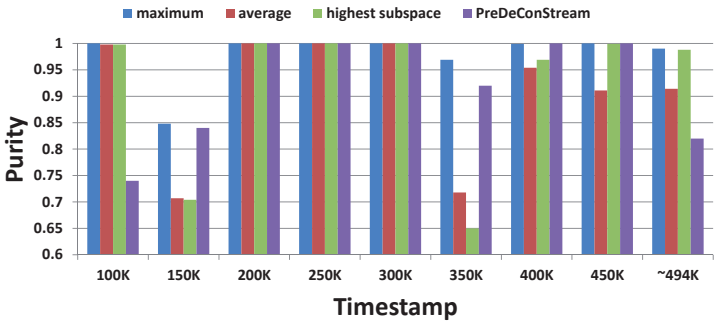


Figure 8.9: Clustering quality of clusters within different subspaces found using SubClusTree compared with that of PreDeConStream (Chapter 5) stream projected clustering algorithm using the Network Intrusion Dataset.

T_s	subspace of *maximum*	subspace of *highest*
100K	{25,28}	{2, 16, 23, 24, 25, 29}
150K	{0, 15, 16, 23,24,28}	{2, 15, 16, 20, 23,24,28}
200K	{15, 16}	{15, 16}
250K	{1, 15, 16}	{1, 15, 16}
300K	{1, 15, 16, 24}	{1, 15, 16, 24}
350K	{18}	{1, 16, 24}
400K	{18, 28}	{15, 16, 17, 18, 23, 28}
450K	{1, 16, 24, 25}	{1, 2, 16, 24, 25}
494K	{2, 24}	{16, 24, 28}

Table 8.1: Delivered subspaces of the *maximum* and *highest-dimensionality subspace* clusterings that are evaluated in Figure 8.9.

of η to 0.5 or 1, we can see a considerable fall in purity. This is due to the fact that many dimensions, that contain noise, have a potential higher than these values. Setting η to greater values than 1.5 also yields a decrease in purity. This is because we exclude some dimensions although they are dense. Figure 8.11(a) shows the achieved average purity for several batch sizes. As we can see, purity is above 80% for batch sizes ranging from 2500 to 30,000 while there is clear fall in the purity for batch size ranges from 20,000 to 50,000. Higher values of the batch size are not suitable for rapidly evolving stream. Because over the course of time, many old instantiated trees representing specific subspaces may lose their relevance and should be deleted. In addition to this, there may be a concept drift that should be detected instantly. Figure 8.11(b), illustrates the attained purity with respect to varying window sizes and horizon $H = 1$. Since we set the decay

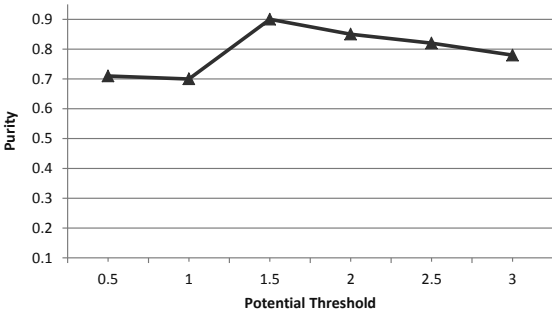
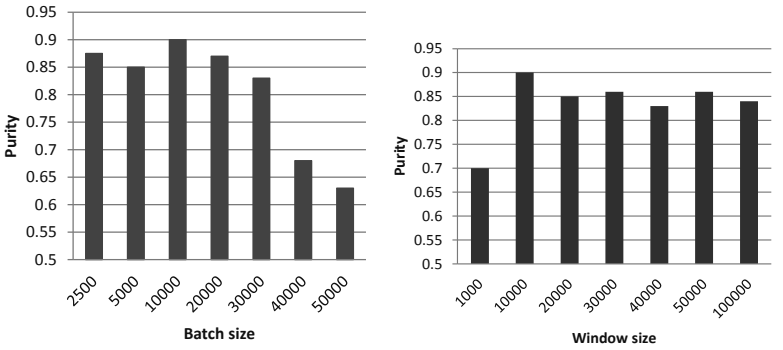


Figure 8.10: Sensitivity to η (Synthetic dataset N200KC3D25R13, $H = 1$, $w = 10,000$)

factor $\lambda = 1$ which lets the algorithm forgets old data faster, the purity reached with a window size = 1000 is considerably low. For all other values of window size, purity is above 80%. The optimal value for the used synthetic generator is proved to be 10,000. That is why we set the standard window size equal to this value. Sensitivity analysis (cf. Figures 8.10 and 8.11) has proved that the SubClusTree is not very sensitive to parameters.

Scalability Test

In the first scalability test, we varied the number of relevant dimensions. The total number of objects was set to N= 200K, the total number of dimensions D = 25 and the number of clusters C = 3. Figure 8.12(a) depicts the results. The average purity raises if we increase the number of relevant dimensions up to half of the total dimensionality. For R = 5 to 9, the proportion of the irrelevant dimensions, is relatively large and thus the achieved purity is 75$\%$. The optimal values range between R = 13 to 21 and we obtain a purity of about 90$\%$. To test the scalability against the number of dimensions, we generated the synthetic data with$N$ = 200K and C = 3. The number of relevant dimensions R was set to 50% of the total number of dimensions. As we can see, from Figure 8.12(b), SubClusTree is highly scalable with dimensionality. For a dataset of dimensionality ranging from 20 to 25 , the average purity is nearly 90$\%$. If we increase the number of dimensions up to 50, the gained average purity is still 80$\%$, which is quite good.



(a) (b)

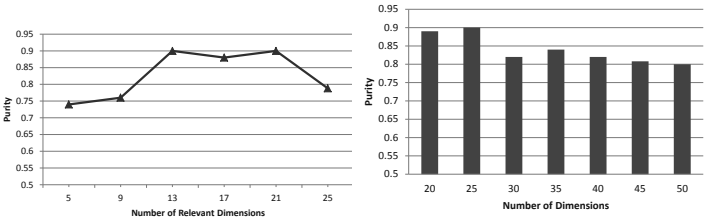
Figure 8.11: (a) Sensitivity to batch size (Synthetic dataset N200KC3D25R13, $H = 1$, $w = 10,000$). (b) Sensitivity to window size (Synthetic dataset N200KC3D25R13, $H = 1$).

Change of The Ground Truth

This experiment was aimed to determine how well SubClusTree can detect the novel concepts. To achieve this, we generated the synthetic data with the standard setting first, i.e. N200KC3D25R13, $H = 1$, $w = 10,000$. After the timestamp 100K, we changed the ground truths of our dataset. In the standard settings, 3 clusters with 13 relevant dimensions were used. We deleted the 3 old clusters and then generated 2 new clusters with 10 relevant dimensions (different than before). Figure 8.13 depicts the achieved average purity with the evolution of time. At timestamp 50K, the algorithm obtained an average purity of 75%. It attained its best possible value 92% at timestamp 100K. At timestamp 110K, SubClusTree noticed the changes in the stream; the subspace trees (that were previously relevant) had a big drop in purity and simultaneously the 1D trees (that were previously irrelevant) had a purity of above 90%. Accordingly, the obtained average purity was about 80%. In the next batch calculations, the trees that became irrelevant were deleted and new relevant candidates were initialized. But the purity increased gradually and reached its highest value at timestamp 200K.

Anytime Capability

Until now, we tested SubClusTree with a constant speed stream. In this experiment, we varied the average number of time steps λ_p between objects and mea-



(a) (b)

Figure 8.12: (a): Scalability with relevant dimensions. (b): Scalability with dimensionality.

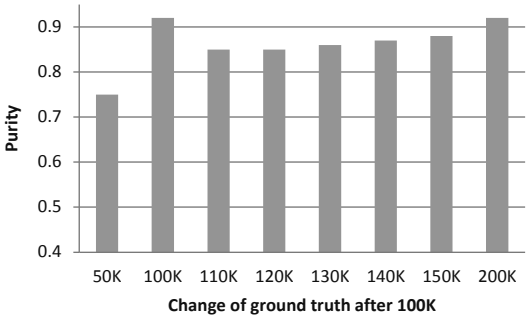


Figure 8.13: Average purity attained by change in ground truth after 100K timestamp.

sured the clustering quality on synthetic dataset. We evaluated the average purity for $\lambda_p = 5$ (fast stream) to 150 (slow stream). From Figure 8.14, we can see that an average interval of 50 steps between objects is very close to the best possible value. The more time we have, the better the clustering quality we achieve. This is completely compatible with the anytime concept [KABS09]. For a very fast stream with $\lambda_p = 5$, we experienced a decrease in the clustering quality. Because in such a case, SubClusTree is only able to insert the objects in 1D trees plus five instantiated subspace trees. We see a little drop in clustering quality at average step 20, this is due to the fact that objects are inserted in randomly selected trees.

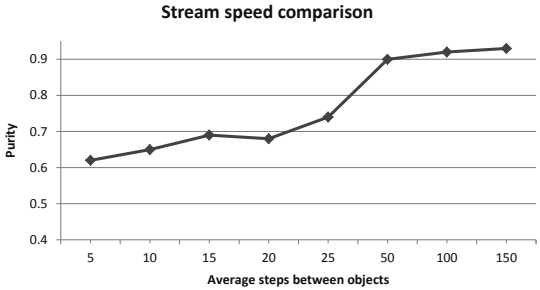


Figure 8.14: Average purity achieved by varying the inter-arrival times of the objects.

8.5 Conclusion

In this chapter, we proposed a novel algorithm for subspace anytime stream clustering called *SubClusTree*. It processes the data objects in a single pass and maintains the microclusters in their relevant subspaces efficiently. It uses a tree structure with a bit-vector to represent a subspace, where the bit-vector keeps track of the dimensions corresponding to the subspace. The tree structure stores finer summaries of data in its leaf nodes and coarser information in its inner nodes. Furthermore, it makes no assumption regarding the number of clusters and can keep a larger number of microclusters. To capture the evolving behavior of the streaming data, it incorporates a decaying mechanism that allows old concepts to expire. SubClusTree uses a smart strategy to bypass the step-by-step method (bottom-up detection of subspace candidates) and can directly jump to higher subspaces. It takes the advantage of flexible grids and joins them randomly to form subspace candidates. It prunes the candidates that do not have enough potential to become a cluster. In an extensive experimental evaluation using synthetic and real datasets, we have shown that it is not very sensitive to the parameters, scalable and has a quite stable clustering quality. It can detect the instant changes in stream and updates the relevant subspaces accordingly. We also varied the average speeds of the stream to test the anytime capability of SubClusTree which was compatible with the anytime concept.

Part IV

Framework and Evaluation Measures for Stream Subspace Clustering

Chapter 9

The *Subspace MOA* Framework

* Recently, the *OpenSubspace* framework was proposed to evaluate and explorate subspace clustering algorithms in WEKA with a rich body of most state of the art subspace clustering algorithms and measures. Parallel to it, MOA (**M**assive **O**nline **A**nalysis) framework was developed also above WEKA to provide algorithms and evaluation methods for mining tasks on evolving data streams over the full space only. Similar to static data, most streaming data sources are becoming high-dimensional, and tracking their evolving clusters is also becoming important and challenging.

In this chapter, we present, to the best of our knowledge, the first subspace clustering evaluation framework over data streams called *Subspace MOA*. Our framework has three phases. In the online phase, users have the possibility to select one of three most famous summarization techniques to form the microclusters. Upon a user request for a final clustering, the regeneration phase constructs the data objects out of the current microclusters. Then, in the offline phase, one of five subspace clustering algorithms can be selected. The framework is supported with a subspace stream generator, a visualization interface and various subspace clustering evaluation measures.

9.1 Motivation

The *OpenSubspace* framework [MAG+09a] was proposed to evaluate and explore subspace clustering algorithms in WEKA [HFH+09] with a rich body of most state

*This chapter has been published in the Proceedings of the 18th International Conference on Database Systems for Advanced Applications, DASFAA 2013 [HKS13].

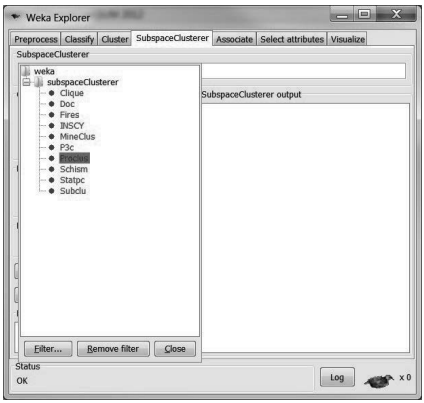


Figure 9.1: A screen shot of the *OpenSubspace* framework.

of the art subspace or projected clustering algorithms and evaluation measures (cf. Figure 9.1).

In this chapter, these algorithms are applied to the streaming cases. Other than static data that do not vary over time, streaming data are given in a different rate and a dynamically-changing pattern, which makes it challenging to analyze its evolving structure and behavior. In streaming scenarios, we also often face limitations on processing time and storage, since a vast amount of continuous data are coming rapidly.

MOA (Massive Online Analysis) framework [BHKP10] was built on experience with both WEKA [HFH+09] and VFML (Very Fast Machine Learning) toolkit [HD03] to support the research in the stream mining area with generators, visualization methods, and interesting evaluation measures. Similar to static data, evolving data streams are also becoming naturally high-dimensional with their existence in multiple applications with many attributes. However, different to subspace clustering algorithms over static data, only few subspace stream clustering algorithms have been developed recently (HPStream [AHWY04], HDDStream [NZP+12] and PreDeConStream [HSGS12] cf. Chapter 5). Such kinds of algorithms are a bit tricky since they have to track all changes of evolving clusters over the streams (splitting, merging, appearance, decaying, moving etc.), by considering the fact the these clusters might exist in all possible subspaces and not only in the full-space. In *Subspace MOA*, users can select any of five subspace clustering algorithms to be the offline part of a subspace clustering algorithm, where one of

three summarization methods for the online part can be also selected. Additionally, users select one of the two implemented standalone projected stream clustering algorithms: PreDeConStream [HSGS12] (cf. Chapter 5) and HDDStream [NZP+12]. Users can also compare and visualize the output of two algorithms of their own choice with the help of multiple subspace and stream evaluation measures.

The remainder of this chapter is organized as follows: Section 9.2 explains the SubspaceMOA framework including the different tabs and processing steps. An extensive evaluation of many combination of the resulted algorithms is discussed in Section 9.3 to show an example on how would the framework be used for highlighting the pros and the contras of different offline algorithms for the dataset at hand. In Section 9.4 we conclude this chapter.

9.2 The *Subspace MOA* Framework

In this section we will discuss the contents of each tab available in *Subspace MOA* framework in details.

9.2.1 The *Setup* tab and the Underlying Algorithmic Model

Figure 9.3(a) shows this tab which offers the possibility of selecting the **data stream input**. *Subspace MOA* offers a synthetic random RBF subspace generator with the possibility of varying multiple parameters of the generated stream and its subspace events. One can vary: the number of dimensions, the number of relevant dimensions (i.e. the number of dimensions of the subspaces that contain the ground truth clusters), the number of the generated clusters, the radius of the generated clusters, the speed of the movement of the generated clusters, the percentage of the allowed overlapping between clusters, and the percentage of noise. Please note that some dimensions of a point could represent a noise within some subspace, while other dimensions of the same point could be a part of a ground truth cluster in other subspace. The generated noise percentage in this case represents a guaranteed noise in all subspaces. *Subspace MOA* gives also the possibility of reading external ARFF files. In this section we will present also the model we are using for applying subspace stream clustering algorithms to efficiently and effectively find subspace clusters within big data. In the experimental part, we will show a comparison between many possible algorithmic scenarios in

details. The idea of stream subspace clustering is intuitive: we cluster the incoming data "live" and save features of the clustering for a defined period of time. We can then approximately reconstruct the data for a given time frame, the more recent the more accurate, and use a classic subspace algorithm to determine a clustering of this frame.

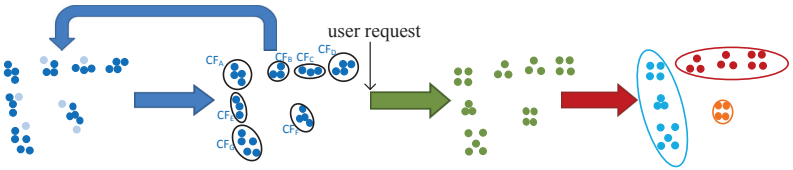


Figure 9.2: The whole process of our model for stream subspace clustering for big data. The blue arrows represent the online phase, the green arrow represents the regeneration phase and the red arrow represents the offline phase.

We use a three-stop approach in this chapter: an online phase, a regeneration phase and then an offline phase. The data stream gets processed by CluStream [AHWY03] or DenStream [CEQZ06] in the online phase, which produces microclusters for the current input data as will be explained later. These clusters are then saved as cluster feature vectors as seen in Figure 9.2. Then, upon some request from the user for a final clustering or after a certain amount of time, we regenerate the points out of the summaries in the regeneration phase. The regenerated data is then forwarded to one of the final five subspace clustering algorithm which produces the final clusters.

The Online Phase: A Stream Clustering Algorithm

In the online phase of our model, a summarization of the data stream points is performed and the resulting microclusters are given by sets of cluster features:

$$CF_A = (N, LS_i, SS_i)$$

which represent the number of points within that microcluster$A$, their linear sum and their squared sum, respectively. One of the three online algorithms (CluStream, DenStream or a basic cluster generation method) is responsible for forming these microclusters, deleting older ones or continuously maintaining the updated ones.

The Regeneration Phase: Gaussians Out of Online Summaries

After reaching a predefined time threshold, we call it here (window size), we compute, normally distributed objects over each each dimension out of the statistics we got from the cluster features of the microclusters (the green arrow in Figure 9.2).

This step is called the regeneration phase, where the clustering features are used to reconstruct an approximation to the original N points, for each microcluster, using Gaussian functions to reconstruct points over each dimension i.

$$c_i = \frac{LS_i}{N}$$

with a radius:

$$r = \sqrt{\frac{SS}{N} - (\frac{LS}{N})^2}$$

where:

$$SS = \frac{1}{d}\sum_{i=1}^{d} SS_i$$

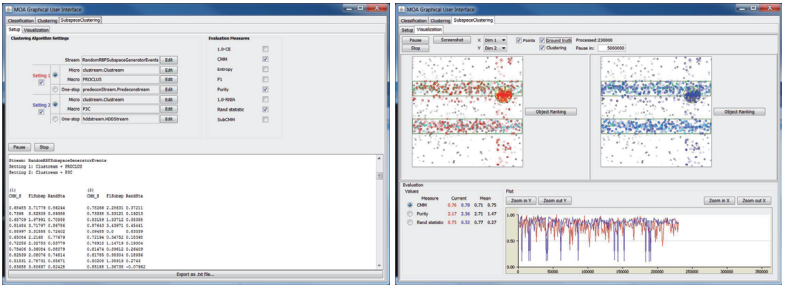
and

$$LS = \frac{1}{d}\sum_{i=1}^{d} LS_i$$

The generated N_A points for each microcluster will be now normally distributed. Thus, they will look a little bit differently distributed than the original distribution (compare the green points in Figure 9.2 with the dark blue ones to the left). Actually, this is the only approximation that we have in our model.

The Offline Phase: A Subspace Clustering Algorithm

The generated N points are then forwarded to one of the five subspace clustering algorithms (the red arrow in Figure 9.2). These are SUBCLU [KKK04], ProClus [AWY+99], Clique [AGGR98], P3C [MSE06] and FIRES [KKRW05]. This results with 15 different combinations of algorithms that can be tested. These algorithms are applied to the streaming cases. Other than static data that do not vary over time, streaming data are given in different rates and dynamically-changing patterns, which makes it challenging to analyze its evolving structure and behavior. In streaming scenarios, we also often face limitations on processing time and storage, since a vast amount of continuous data are coming rapidly.



(a) (b)

Figure 9.3: Subspace MOA screen shots of (a) The setup tab, (b) The visualization tab.

9.2.2 The *Visualization* Tab

Figure 9.3(b)) depicts the output that can be seen under this tab. We have adapted the most famous offline subspace clustering evaluation measures (CE [PM06], Entropy [SZ05], F1 [AKMS08], RNIA [PM06]), to the streaming scenario. Additionally, we have implemented our novel **SubCMM** measure (cf. Chapter 10). The user has the possibility to select the evaluation frequency, the window size is then set accordingly, and the evaluation measure is applied over the found clusters when compared against the ground truth clusters within that window. The output of these evaluation measures is delivered to the user according to the MOA conventions in three ways: (a) in a **textual form**, where summarization values are printed gradually in the output panel under the "Setup" tab as the stream evolves, (b) in a **numerical form**, where recent values of all measures are printed instantly under the "Visualization" tab, and (c) in a **plotted** form of the selected measure from the recent values. The evolving of the final clustering of the selected subspace clustering algorithms as well as the evolving of the ground truth stream is visualized in a two dimensional representation. Users can select any pair of dimensions to visualize the evolving ground truth as well as the resulted clustering. Different to MOA, Subspace MOA is able to visualize and get the quality measures of arbitrarily shaped clusters.

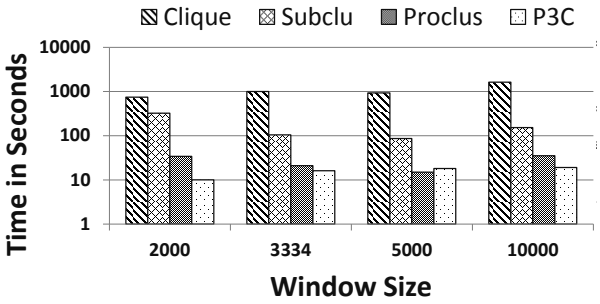


Figure 9.4: The performance of the four offline algorithms after using CluStream in the online phase when changing the window size (the batch size).

9.3 Experimental Evaluation

In this section, we test the Subspace MOA framework using a combination of CluStream and SUBCLU on the Network Intrusion Dataset [Dat99]. To compare the offline algorithms, we will test a variety of other macro algorithms according to accuracy and performance. All calculations were done on a AMD FX 8-core clocked at 4 GHz with 8 GB RAM. Some of the following results appeared in the evaluation work: [HS14].

9.3.1 Running Time Results

To evaluate the performance we have set the CluStream with a maximum of 30 microclusters, and compared four different window sizes in the range from 2000 to 10000 for four different macro clustering algorithms while keeping the number of the overall processed objects at a steady 10000. The parameter settings of the different algorithms were set as suggested in their original papers in the default values within Subspace MOA. These are $\xi = 10$ and $\tau = 0.01$ for CLIQUE, $\epsilon = 0.002$ and $m = 5$ for SubClu, $c = 5, d = 2$ for PROCLUS, and $p = 10, \chi^2 = 0.001$ for P3C.

Note that in Figure 9.4 the logarithmic scaling of the runtime in seconds as CLIQUE was too slow to have all values in one figure using an arithmetic scaling.

CLIQUE is in every aspect the slowest algorithm even at its best settings. It needed around 27 minutes to process all the 10000 objects in one window. SUBCLU is vastly more efficient. for the different window sizes, it took between

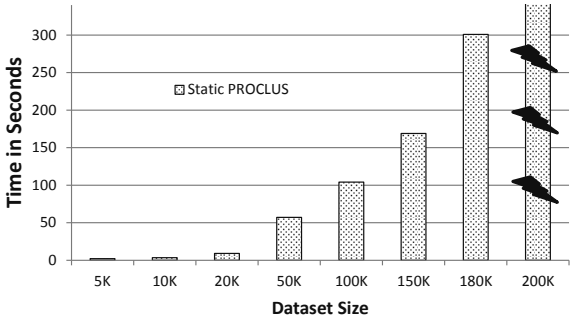


Figure 9.5: The runtime performance of a *static* subspace clustering algorithm: PROCLUS. Beginning from a sub-dataset size of 200K objects, the algorithm failed to successfully finish the running.

80 seconds and 5 minutes in total. Interestingly, a window size of 2000, and thus doing 5 computations of each, yields the worst results while two computation circles of 5000 objects each are results with a better total running time. PROCLUS and P3C are both extremely fast when compared to SUBCLU with 35 seconds for the worst-case window. While PROCLUS shows the same preference for a 5000-object window size as SUBCLU, P3C prefers smaller windows. To observe the huge improvement our model brings when compared to the static subspace clustering algorithms, we tried to apply the Network Intrusion Dataset [Dat99] over the static PROCLUS algorithm. We have tried first to run the PROCLUS over the whole dataset size with 1 GB memory allocated for the algorithms' heap. As it was crashing, we decided to try smaller versions of the dataset, by getting the first ones as they appear in the dataset. As shown in Figure 9.5, the exponential increase of the runtime is obvious as the size of the dataset increases. The algorithms started to crash when trying a sub-dataset of size $200K$. Additionally, the runtime improvement that our algorithmic model causes over PROCLUS is obvious when trying any window size (cf. Figure 9.4).

9.3.2 Accuracy Results

In the context of highlighting the pros and the contras of each algorithm combination using Subspace MOA, a total of three different evaluation measures are used in this section. The $F1$ measure, which gives a overview how well hidden clusters are represented by the output of the algorithm, RNIA [PM06], which

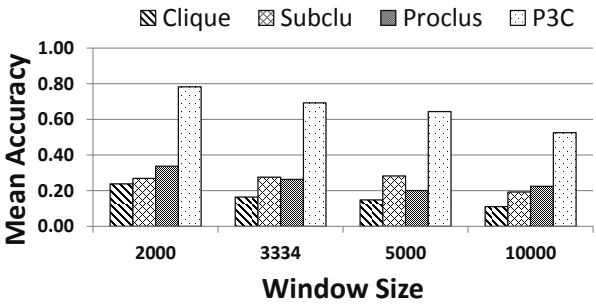


Figure 9.6: The averaged accuracy of the four offline algorithms when using CluStream in the online phase when changing the window size (the batch size).

measures how well hidden subobjects are covered by already found objects, and CE [PM06], which works similar to RNIA but additionally evaluates if a cluster is split up into several smaller clusters. From now on, when talking about RNIA or CE measures, we mean $1-CE$ and $1-RNIA$, so the closer the measure to 1, the better. This is to make the results comparable with those of $F1$ measure. For the previous settings of the performance evaluation, we averaged for each algorithm the three accuracy evaluation measures. Figure 9.6 depicts the gained results as the batch size (window size) changes. As expected, the accuracy of almost all algorithms increases when the window size increases, with one exception with SubClu which is fluctuating a bit. The reason of this improvement of the accuracy is the fact that considering more data at a time, gives each algorithm more possibility to find even more hidden clusters. Another observation, is that nearly all of the algorithms who performed well w.r.t. the running time, are also accurate. The slow ones are also delivering additionally bad results. This makes P3C a winner algorithm when considering the running time and accuracy.

Going over the different settings of CluStream and SUBCLU, we first check for different values of ϵ how would the performance be affected. Just as in the performance section, we use 30 as the maximum number of microclusters for CluStream, 2000 as the window size and $m = 5$ for SUBCLU.

As can be seen in Figure 9.7, the $F1$ measure start to drop after $\epsilon = 0.002$. Prior to this value, smaller settings meant that there are more clusters but of smaller sizes resulting in the balance between precision and recall. After this, a bit of precision is lost, resulting in the 0.1 worse measure. Starting from a smaller $\epsilon = 0.0015$, both RNIA and CE have a maximum with RNIA falling a bit

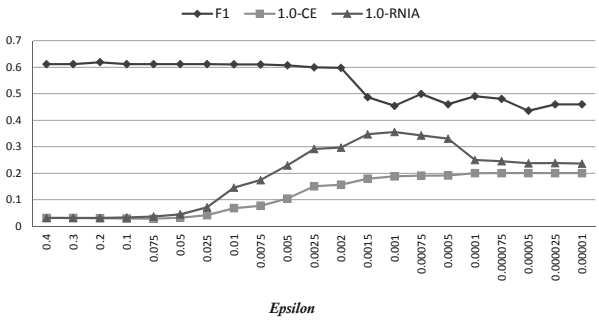


Figure 9.7: Different accuracy measures of the SUBCLU algorithm after using CluStream in the online phase when changing ϵ (the neighborhood parameter).

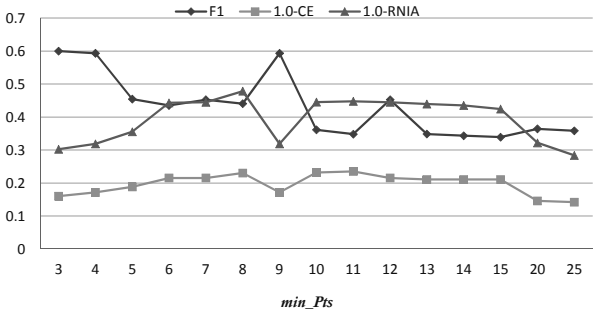


Figure 9.8: Different accuracy measures of the SUBCLU algorithm after using CluStream in the online phase when changing m (the minimum number of points needed in the ϵ-neighborhood for an object to become core [KKK04]).

after $\epsilon = 0.0001$. It looks like the algorithm did not find too many objects after this part, however the found ones are clustered with little excessive clusters. Another parameter to check is the minimum points m, found in Figure 9.8. For this benchmark, $\epsilon = 0.001$ was used. $m = 9$ seems to be an interesting point, resulting in a spike from both $F1$ and RNIA in opposite directions. We could assume this was a threshold for adding "bad" objects to a cluster without enabling DBSCAN to connect the cluster to an existing "good" one.

Overall, this setting seems to have lower impact on the accuracy of SUBCLU than the ϵ parameter. As these settings are mainly used for DBSCAN this is not unsurprising but still nice to confirm.

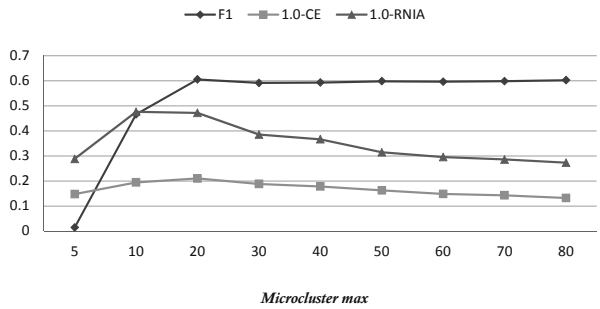


Figure 9.9: Different accuracy measures of the SUBCLU algorithm after using CluStream in the online phase when changing the maximum allowed number of microclusters within CluStream.

We will now check how would varying the maximum amount of microclusters within CluStream affect the quality of the results. Figure 9.9 depicts the results, as expected, a certain minimum amount of microclusters has to be present to achieve good results. After this threshold, which seems to be around 20, the results do not change too much but slowly get worse. This is true especially for the RNIA measure.

9.4 Conclusion

In the chapter, we introduced the *Subspace MOA* framework as a first open source framework used for embedding, evaluating and visualizing the output of subspace stream clustering algorithms. As there exists only two standalone projected stream clustering algorithms, we have used a special online-reconstruction-offline model that gives the user the possibility of combining their own algorithm out of 3 online summarization methods and 5 offline subspace clustering techniques. In addition to enabling the reading of external datasets, the framework is supported with a subspace stream generator. Multiple state-of-the-art subspace and stream clustering evaluation measures are also included. In the experimental evaluation we have given an example on how would the framework offer researchers the possibilities to detect pros and contras of different subspace clustering algorithms when applied in the streaming scenario. Additionally, the suitable online-offline algorithmic combination for a certain dataset can be decided. This is all done in a user friendly interface that is in line with the MOA framework style. Subspace

MOA can be found at `http://dme.rwth-aachen.de/en/subspacemoa`.

Chapter 10

Subspace Cluster Mapping Measure (SubCMM)

* Available stream clustering evaluation measures care only about the errors of the *full-space* clustering but not the quality of subspace stream clustering. On the other hand, existing subspace clustering evaluation measures are mainly designed for static data, and cannot reflect the quality of subspace clustering algorithms over the evolving data streams.

In this chapter we propose, to the best of our knowledge, the first subspace clustering measure that is designed for streaming data, called *SubCMM*: **Sub**space **C**luster **M**apping **M**easure. *SubCMM* is an effective external evaluation measure for stream subspace clustering that is able to handle errors caused by emerging, moving, or splitting subspace clusters. Additionally, we extensively compare our new measure against state-of-the-art full-space stream clustering evaluation measures. The experimental evaluation, that is performed using the Subspace MOA framework, depicts the ability of *SubCMM* to reflect different changes happening in the subspaces of the evolving stream.

10.1 Motivation

Data sources are increasingly generating more and more data and the huge advances of data sensing systems resulted in cheap means for satisfying the eagerness for collecting data with a high number of attributes. The big size of the data together with its high dimensionality motivated the research in the area of

*This chapter has been published in the Journal of Intelligent Information Systems (JIIS) and presented at the PAKDD 2013 QIMIE workshop [HKCS13].

high dimensional data mining and exploration. Data stream is a form of data that continuously and endlessly evolves reflecting the current status of collected values.

Evaluating full-space stream clustering algorithms, can be done mainly by assessing: (a) the **efficiency** represented by the runtime, the memory usage or the number of microclusters processed by the algorithm when mining the stream with different speeds, and (b) the **effectiveness** represented by the quality of the resulted clusters which mainly compares the found evolving clusters to the ground truth ones. Most of these were inherited from the offline clustering world, only one was mainly designed for streaming algorithm (cf. CMM [KKJ+11]).

In many applications of streaming data, objects are described by using multiple dimensions (e.g. the Network Intrusion Dataset [Dat99] has 42 dimensions). For such kinds of data with higher dimensions, distances grow more and more alike due to an effect termed *curse of dimensionality* [BGRS99]. The latter fact motivated the research in the domain of *subspace* and *projected clustering* in the last decade which resulted in an established research area for static data.

In parallel to developing these static data subspace clustering algorithms, a group of measures for evaluating the clustering quality of offline subspace clustering algorithms were established. Additionally, other measures were inherited from traditional full-space clustering world (e.g. RNIA, CE [PM06], Entropy [SZ05], Accuracy [BZ07] and F1 [AKMS08]). For streaming data on the other hand, although a considerable research has tackled the full-space clustering, relatively limited work has dealt with subspace clustering. HPStream [AHWY04], PreDeConStream [HSGS12] (cf. Chapter 5), HDDStream [NZP+12], SiblingList [PL07] and SubClusTree [HKSS14] (cf. Chapter 8) are so far the only works that have been done on projected/subspace stream clustering.

Almost all of the above mentioned algorithms have used the clustering purity [ZK04] as the only measure for assessing the clustering quality (except PreDeConStream and SubClusTree cf. Sections 5.5 and 8.4). The purity was not mainly designed for subspace stream clustering, and does not reflect the cases when clusters hidden in some subspaces are completely not discovered.

Stream clustering evaluation measure is not a part of the clustering algorithm itself, thus it will not negatively affect the efficiency of the clustering algorithm. The evaluation of a clustering algorithm is a step that is done separately to assess the clustering algorithm, and does not need to be performed always when the algorithm runs. Consequently, when designing the stream clustering measure, a

special care must be taken on the ability of the measure of reflecting the real current distribution of the stream, also when the stream speed is high. Nevertheless, the design of an efficient measure is an option that is always welcomed.

In this chapter we propose, to the best of our knowledge, the first subspace clustering measure that is designed for streaming data, called *SubCMM*: **Sub**space **C**luster **M**apping **M**easure. *SubCMM* is an effective evaluation measure for stream subspace clustering that is able to handle errors caused by emerging, moving, or splitting subspace clusters. Additionally, we propose a novel method for using available *offline* subspace clustering measures for data streams within the *Subspace MOA* framework [HKS13] (cf. Chapter 9).

The remainder of this chapter is organized as follows: Section 10.2 gives a short overview of the related work from different neighboring areas to full-space and subspace stream clustering algorithms as well as the measures used there. Our SubCMM measure is introduced in Section 10.3. The suggested measure is then thoroughly evaluated using the Subspace MOA framework in Section 10.4. Finally we conclude the chapter in Section 10.5.

10.2 Related Work

In this section, we list the related work from three areas: subspace clustering measures for static data, full-space stream clustering measures, and available subspace stream clustering and measures. Finally we will detail CMM [KKJ+11], the most related stream clustering evaluation measure.

10.2.1 Subspace Clustering Measures for Static Data

SubClu [KKK04] is a subspace clustering algorithm that uses the DBSCAN [EKSX96] clustering model of density connected sets. PreDeCon [BKKK04] is a projected clustering algorithm which adapts the concept of density-based clustering [EKSX96] and the preference weighted neighborhood contains at least μ points. IncPreDeCon [KKNZ10] is an incremental version of the algorithm PreDeCon [BKKK04] designed to handle accumulating data.

Evaluating the quality of the clustering delivered by the above algorithms was performed using a set of measures, which can also be categorized according to [MGAS09] depending on the required information about the ground truth clusters into two categories:

1. **Object-based measures:** where only information on which objects should be grouped together to form a cluster are used. Examples are: **entropy** [SZ05] which measures the homogeneity of the found clusters with respect to the ground truth clusters, **F**1[AKMS08] which evaluates how well the ground truth clusters are represented and **accuracy** [BZ07].

2. **Object-based and subspace-based measures:** where information on objects as well as their relevant dimensions (i.e. the subspaces where they belong to) are used. Examples are: (a)**RNIA** [PM06] (Relative Non Intersecting Area) which measures to which extent the ground truth subobjects are covered by the found subobjects and (b) **CE** [PM06] (Clustering Error) which is an advanced version of RNIA and differs in that it maps each found cluster to at most one ground truth cluster and also each ground truth cluster to at most one found cluster.

10.2.2 Full-space Clustering Measures for Streaming Data

There is a rich body of literature on stream clustering. Convex stream clustering approaches are based on a k-center clustering [AHWY03, HMS09]. Detecting clusters of arbitrary shapes in streaming data has been proposed using kernels [JZC06], fractal dimensions [LC08] and density-based clustering [CEQZ06, CT07]. Another line of research considers the anytime clustering with the existence of outliers [HKS11].

To reflect the quality of the *full-space* clustering algorithm, many evaluation measures are used. Some of those are inherited from the offline clustering world (cf. for instance: **SSQ** [HKP06], **Silhouette Coefficients** [KR90] and **purity** [ZK04]). Other measures were developed specifically for assessing the quality of full space stream clustering algorithms like **CMM** [KKJ$^+$11] (cf. Section 10.2.4).

10.2.3 Subspace Clustering Measures for Streaming Data

Sibling Tree [PL07] is a grid-based *subspace* clustering algorithm where the streaming distribution statistics is monitored by a list of grid-cells. Once a grid-cell is dense, the tree grows in that cell in order to trace any possible higher dimensional cluster. HPStream [AHWY04] is a k-means-based *projected* clustering algorithm for high dimensional data stream. PreDeConStream [HSGS12] (chapter 5) and HDDStream [NZP$^+$12] are recent density-based projected stream

clustering algorithms that were developed developed upon PreDeCon [BKKK04] in the offline phase.

Almost all of the above mentioned subspace stream clustering algorithms have used the clustering **purity** [ZK04] as the only measure for assessing the clustering quality. The average purity of the clusters in a subspace sub is defined as: Purity(sub) $= \frac{\sum_{i=1}^{C_{sub}} \frac{|C_{i,sub}^{dom}|}{|C_{i,sub}|}}{C_{sub}}$ where $|C_{i,sub}^{dom}|$ denotes the number of points with the dominant class label in cluster i within the subspace sub and $|C_{i,sub}|$ denotes the number of the points in the cluster i within the subspace sub. C_{sub} represents the number of clusters within the subspace sub. The total purity of a subspace clustering algorithm is then calculated by averaging the purities of all subspaces that contain clusters.

Although the purity has proved to be popular and good when used with full-space stream clustering, it was not mainly designed for subspace stream clustering, and does not reflect the cases when clusters hidden in some subspaces are completely not discovered as our measure SubCMM does. Additionally, an another difference to our SubCMM: because of its property of neglecting the shape of the ground truth, errors occurring on the borders of detected micro-clusters are not correctly punished due to the fast change of the shape or the position of the cluster. Although both SubCMM and purity process the output data batch-wise, the purity use the sliding window concept there, while SubCMM use the damped window concept with an aging function. Thus SubCMM is more streaming-friendly, by punishing recent misplaced points more than outdated ones.

10.2.4 Review: CMM

We will review CMM [KKJ+11] (Cluster Mapping Measure) separately here, since it is the only stream clustering measure that was designed for streaming applications, and because it strongly related to our proposed measure: SubCMM.

Static measures like structural and ground-truth-based measures, cannot correctly reflect errors attributable to emerging, splitting, or moving clusters. These situations are inherent to the streaming context due to the dynamic changes in the data distribution. CMM is an external evaluation measure, that fills the gap that all static measures had when being applied in the streaming scenario, and punishes the errors caused by the above three changes using its three phases:

First, each found cluster is assigned to one of the ground truth clusters based

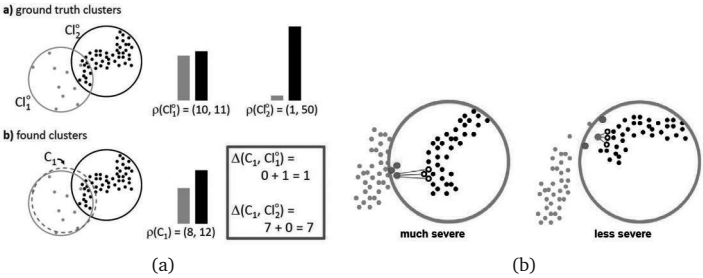


Figure 10.1: [KKJ+11] CMM: (a) The mapping phase of the found cluster to a ground truth cluster, (b) Different penalties for two clusterings having the same accuracy.

on class distribution in each cluster. In Figure 10.1(a), a plain circle represents a ground cluster, and a dashed circle means a predicted cluster. Each dot is a data point having its class label expressed by colors. Class frequencies are counted for each cluster, and each prediction cluster is mapped to a ground truth cluster that has the most similar class distribution. For Figure 10.1(a), the found cluster is mapped to the gray circle ground truth cluster.

Second, the penalty for every incorrectly predicted point is calculated. In Figure 10.1(a), it can be seen that a lot of black points are included in the found cluster, which are incorrectly clustered, and some of the gray points are excluded in the cluster even if they are not noises. These points are "fault objects" and give they are penalized like this: $pen(o, C_i) = con(o, Cl(o)).(1-con(o, map(C_i)))$ where C_i is a prediction cluster to which the object o belongs to, $Cl(o)$ is a ground truth cluster (hidden cluster) representing the original class label of o, and $map(C_i)$ is the hidden cluster on which C_i is mapped through the cluster mapping phase. The two clusters in Figure 10.1(b) have the same accuracy, but it looks obvious that the left clustering has a bigger problem. If a fault object is closely connected to its hidden cluster, then the error becomes much severe since it was meant to be easily clustered. On the other hand, if the object has high connectivity to the found cluster, CMM allows a low penalty since it was hard to be clustered correctly. The connectivity $con(o, C)$ from an object to a cluster is exploiting the average k-neighborhood distance.

Third, derive a final CMM value by summing up all the penalties weighted over its own lifespan: $CMM(R, H) = 1 - \frac{\sum_{o \in F} w(o).pen(o,R)}{\sum_{o \in O} w(o).con(o,Cl(o))}$ Where R: represents the found clusters, H: represents the ground truth (hidden) clusters, O: is the

set of objects o, F: is fault set, and $w(o)$: is the weight of o. When designing our measures: SubCMM, we have followed a similar three-phase concept as the one of CMM, with carefully considering also the punishment of wrongly clustered objects from clusters with a "strong connectivity" to the subspace.

10.3 *SubCMM*: Subspace Cluster Mapping Measure

We adopted important concepts of CMM (cf. Section 10.2.4) and revised its internal structure to develop a novel evaluation measure for subspace clusterings. The motivation for having a special subspace version of CMM becomes clear when using CMM directly for subspace clustering scenarios. Consider the matrix representation of data in Figure 10.2, where columns represent the objects and rows represent the attributes. Thus, each object is represented by a column, where its lines represent the attributes of this object. Assume that neighboring columns represent neighboring objects. Each circle is an attribute value of an object and the color denotes its class label (gray means noise). Blue, red, purple and orange subspace colors represent ground truth classes and the green dashed rectangle represents the found cluster of some stream clustering algorithm. In Figure 10.2(a), the found cluster is delivered by a full-space stream clustering algorithm, and thus it contains only complete columns in the matrix representation. CMM would not be able to map the found cluster C to the class blue since no obvious single class label for each object can be found. Additionally, a subspace stream clustering algorithm would deliver clusters that look like C in Figure 10.2(b). Here, clusters could contain an arbitrary number of rows. Again, data objects in different clusters are defined in different spaces so we cannot simply count objects to compute class distributions as in CMM. We propose checking the class label of each attribute value (represented here by circle), instead of objects, we call it: Subobjects e_{ij}. Thus, in the matrix representations in Figure 10.2(b), it seems reasonable to assign the found cluster to class blue, since it contains 13 blue circles, one red circle and one noise circle. A similar discussion was mentioned in [PM06], to define the distance between subspace clusters.

Thus, the penalty calculations in current CMM should be changed according to the revised clustering mapping phase. As we construct class distributions in a cluster in the matrix-element-wise way, the fault set consists of fault *matrix elements*, and a fault object o in $pen(o, C_i)$ is to be replaced with a fault matrix element e_{ij}, which is the $j\text{-}th$ subobject of $i\text{-}th$ object (cf. Figure 10.2(b)).

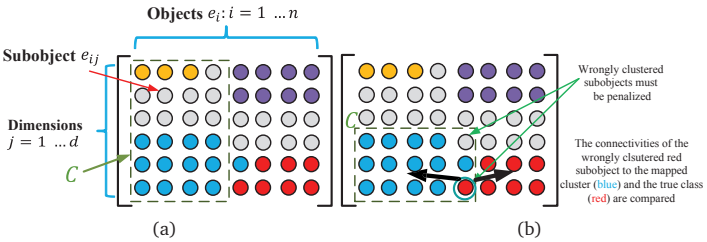


Figure 10.2: (a) Full-space clustering and CMM over the matrix representation of subobjects, (b) Subspace clustering idea using SubCMM and penalizing fault subobjects

Thus the penalty for each wrongly clustered subobject e_{ij} can be calculated as: $pen(e_{ij}, C) = con(e_{ij}, Cl(e_{ij})).(1 - con(e_{ij}, map(C))$. To calculate the penalty in this fashion, we have to define the connectivity between a subobject e_{ij} and a subspace cluster C. In CMM, the connectivity is based on average k-neighborhood distance, which computes Euclidean distance between two data objects and only the difference between attribute values of a same dimension is needed. In the SubCMM, we consider additionally the distance between different dimensions: $con(e_{ij}, C) = subcon(e_{ij}, C).objcon(e_{ij}, C)$ where $subcon(e_{ij}, C)$, the subspace connectivity, represents how much e_{ij} is connected to the subspace of C, and $objcon(e_{ij}, C)$, the object connectivity, means how much e_{ij} is connected to the (sub)objects of C. We define the object connectivity as:

$$objcon(e_{ij}, C) = \begin{cases} 1 & \text{if } [e_{ij} \in C] \text{ or if } [e_{ij} \notin C] \text{ AND} \\ & [knhObjDis^S(e_{ij}, C) < knhObjDis^S(C)] \\ 0 & \text{if } C = null \\ \frac{knhObjDis^S(C)}{knhObjDis^S(e_{ij}, C)} & \text{else} \end{cases}$$

where $knhObjDis^S(e_{ij}, C)$ is the average k-neighborhood distance from e_{ij} to the subobjects in C within the subspace S, and $knhDis^S(C)$ is the average k-neighborhood distance between objects in C within S. The subspace connectivity $subcon(e_{ij}, C)$ is similarly defined as:

$$subcon(e_{ij}, C) = \begin{cases} 1 & \text{if } [j \in S] \text{ or if } [j \notin S] \text{ AND} \\ & [knhDimDis^{e_{ij}}(j, C) < knhDimDis^{e_{ij}}(C)] \\ 0 & \text{if } S \text{ contains no clusters} \\ \frac{knhDimDis^{e_{ij}}(C)}{knhDimDis^{e_{ij}}(j, C)} & \text{else} \end{cases}$$

where: $knhDimDis^{e_{ij}}(j, C)$ is the average k-neighborhood distance from the vector $v_j = [e_{aj}]$ where $a \in C$ to all the vectors $v_l = [e_{al}]$ where $a \in C$ and all $l \in S$ constructed from the objects of C defined in S.
And $knhDimDis^{e_{ij}}(C)$ is the average k-neighborhood distance between vectors v_l constructed from C as above. One can regard this as performing the same procedure of calculating object connectivity on a transposed data matrix. Finally, we have to compute the final SubCMM value with the revised penalties. In this phase, we can just follow the CMM, but the fault object o must be a fault matrix element e_{ij}, and the weights of $e_{ij}s$ are equal when they belong to a same object.

$$SubCMM(R, H) = 1 - \frac{\sum_{e_{ij} \in F} w(i) \cdot pen(e_{ij}, R)}{\sum_{i \in DB} w(i) \sum_{j \in D} con(e_{ij}, Cl(e_{ij}))}$$

10.4 Experimental Evaluation

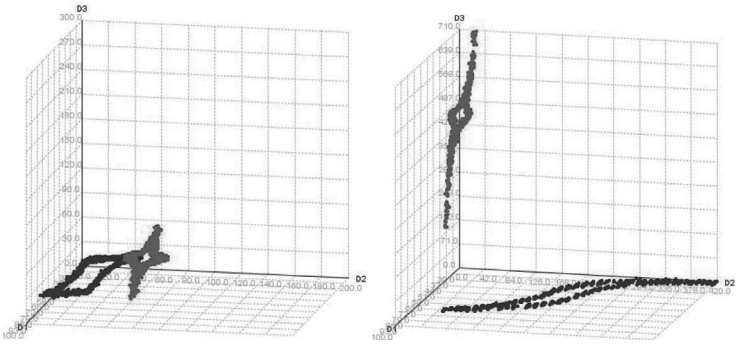
To test the performance of the suggested subspace stream clustering measures, we have used Subspace MOA [HKS13] (cf. Chapter 9) as the testing framework. The remaining parts of this section are organized as follows. Section 10.4.1 lists the datasets used for the evaluation. Section 10.4.2 discusses the stream subspace clustering algorithms used for the evaluation; these algorithms could be a combination of online and offline algorithms, or stand-alone ones. Section 10.4.3 discusses the different parameter settings used when evaluating the algorithms. Finally, Section 10.4.4 presents the evaluation results.

10.4.1 datasets:

We have used two synthetic datasets for the evaluation of our SubCMM: the RBF subspace stream and the $SynStream3D$ dataset.

RBF subspace stream generator:

This dataset was generated using the RBF generator discussed in Section 9.2.



(a) (b)

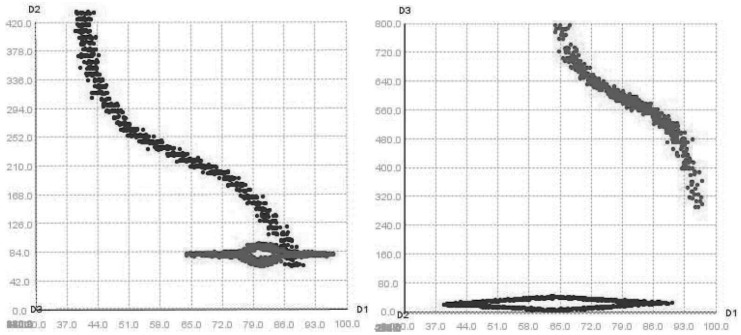
Figure 10.3: An example of a change in the *SynStream3D* streaming dataset (cf. Section 10.4.1): (a) before the change: the two clusters are detected, (b) after the stream evolves, changes: Dimension $D3$ becomes irrelevant for the red cluster and Dimension $D2$ becomes irrelevant of the blue cluster. A full-space stream clustering algorithm will assign all points after the change as outliers, while, in fact, the red points are forming a cluster when considering the Subspace: $(D1, D2)$ (cf. Figure 10.4(a)), and the blue points are forming a cluster when considering the Subspace: $(D1, D3)$ (cf. Figure 10.4(b)).

SynStream3D:

This synthetic dataset was used in Section 5.5.1, we repeat the discussion here for completeness. It consists of 3-dimensional 4000 objects without noise that form at the beginning two arbitrarily shaped clusters over full space (cf. Figure 10.3(a)). After some time, the data stream evolves so that for each cluster different dimensions (one different dimension on each of both clusters) become irrelevant (cf. Figures 10.3(b) and 10.4).

10.4.2 Algorithms Compared

The algorithms that are used in the evaluation part to study the effect over the available measures are some algorithm combinations and a single standalone algorithm. The algorithm combinations are mixed as explain in Section 9.2.1, where the first part represents the online part, whereas the second one (the offline) represents the subspace/projected algorithm. These are: CluStream+PROCLUS [AHWY03] and [AWY+99] for experiments in Figures 10.5-10.7. CluStream+SubClu [AHWY03] and [KKK04] in Figure 10.8.



(a) (b)

Figure 10.4: Two projections of the *SynStream3D* dataset after the change (cf. Figure 10.3(b)) above: (a) shows that the red points are forming a cluster in the Subspace: $(D1, D2)$ while the blue points are noise (mind the scaling on $D2$), (b) shows that the blue points are forming a cluster in the Subspace: $(D1, D3)$ while the red points are noise (mind the scaling on $D3$).

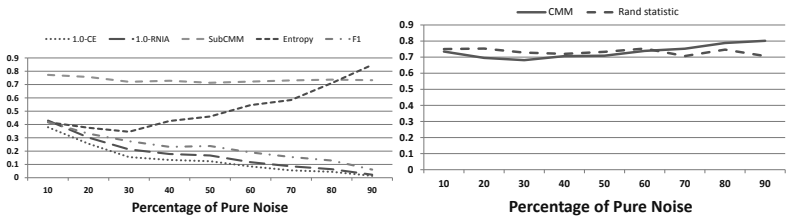
DenStream+PROCLUS [CEQZ06] and [AWY+99] in Figure 10.10. DenStream+P3C [CEQZ06] and [MSE06] in Figure 10.11. DenStream+SubClu [CEQZ06] and [KKK04] in Figure 10.9. In Figures 10.13 and 10.14, we have used the stand-alone stream projected clustering algorithm PreDeConStream (cf. Chapter 5) [HSGS12].

10.4.3 Parameter Settings

We compared the performance of SubCMM, RNIA, CE, Entropy and F1 as representatives of subspace stream clustering measures against the performance of CMM, Rand statistic and the Purity as representatives of full-space stream clustering measures. In all of the following experiments, the RBF subspace stream and algorithm parameter settings, unless otherwise mentioned, or unless that specific parameter is being varied in the experiment, are: number of stream attributes$= 12$, number of attributes of generated clusters$= 4$, number of generated clusters$= 5$, noise level$=10\%$, speed of movement of clusters$=0.01$ per 200 points (which reflects the evolving speed of the concept drift of the stream), the evaluation frequency$= 1000$, and the decaying threshold$= 0.1$.

10.4.4 Evaluation Results

Figure 10.5 compares the performance of subspace stream clustering measures (left) against full-space clustering algorithms when varying the pure noise percentage around the generated ground truth clusters. Apparently, most subspace measures are sensitive to the increase of noise, different to the full-space ones. The stable high value that full-space measures give, is due to the clusters which are accidentally created out of the combination of clusters generated in the lower dimensions. Even for those clusters, when using the full-space measures, the quality does not decrease as in the subspace measures. Figure 10.6 shows the performance of both subspace and full-space measures when varying the number of generated ground truth clusters. Here, the expected effect is a decreasing of the quality as the number of clusters increases. Again, full-space measures are relatively stable, while most subspace measures are sensitive.

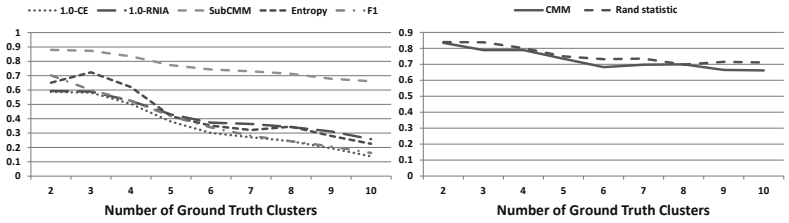


(a) (b)

Figure 10.5: Clustering quality of a subspace stream clustering algorithm when varying the noise level using: (a) Subspace measures, (b) Full-space measures.

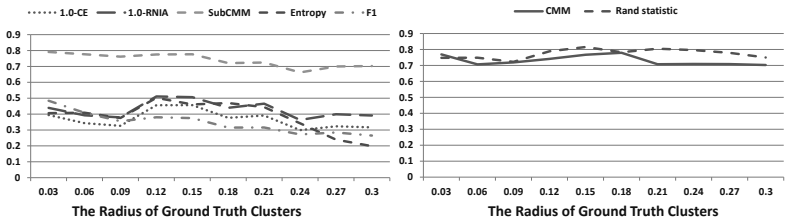
Figure 10.7 depicts the quality of subspace and full-space measures when varying the radius of the generated clusters. Again, the expected change here is a quality decrease as the radius increases. This is clear to see with most subspace measures, while only a slightly decrease can be seen on the full-space measures. The latter decrease, is due to the higher density of the noisy points around the accidentally generated clusters in the full space. This noise might wrongly be added to the clusters in the full-space, and only this noise is punished by full-space measures and not the noise in lower dimensions.

Figure 10.8 depicts the effect over the evaluation measures when varying the number of relevant dimensions. It can be seen that most of the measures, including SubCMM, reach a maximum when the algorithm is tuned such that the



(a) (b)

Figure 10.6: Clustering quality of a subspace stream clustering algorithm when varying the number of clusters using: (a) Subspace measures, (b) Full-space measures.



(a) (b)

Figure 10.7: Clustering quality of a subspace stream clustering algorithm when varying the radius of clusters using: (a) Subspace measures, (b) Full-space measures.

number of relevant dimensions equals to 2. This is exactly as the ground truth states in most of the time of the stream (cf. Figure (10.4)).

Figure 10.9 shows a similar effect of SubCMM when using another algorithm that is based on a density-based clustering, while some other measures wrongly show their maximum when the number of relevant dimensions equals to three.

Figure 10.10 shows the reaction of SubCMM together with the used full-space and subspace measures when varying the number of relevant dimensions over the 12-D RBF subspace dataset. In contrast to the previous two figures, here we have the ground truth containing the clusters in the full space. Thus, the expected output should be maximized close to the 10. This is reflected by SubCMM and most of the evaluation measures except for the CMM and the F1 measures. It should be noted that the generally low values of the measures are due to the non-perfect parameter setting of both the offline and the online parts of the algorithm.

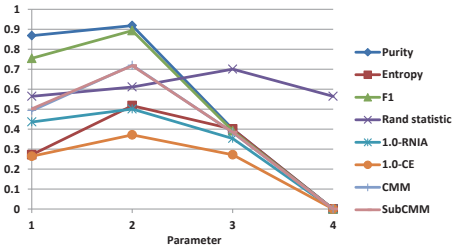


Figure 10.8: Evaluating the CluStream+SubClu combination algorithm with different full-space and subspace evaluation measures against SubCMM using the $SynStream3D$ dataset, when varying the number of relevant dimensions (Parameter= number of relevant dimensions).

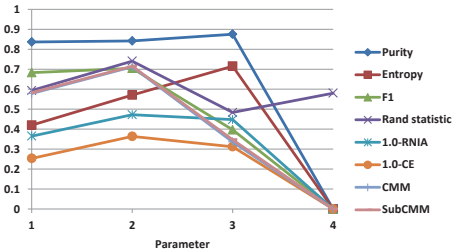


Figure 10.9: Evaluating the DenStream+SubClu combination algorithm with different full-space and subspace evaluation measures against SubCMM using the $SynStream3D$ dataset, when varying the number of relevant dimensions (Parameter= number of relevant dimensions).

Figures 10.11 and 10.12 show the effect of the studied evaluation measures together with SubCMM when varying the evaluation frequency when using the $SynStream3D$ over the DenStream+P3C and the DenStream+PROCLUS combination algorithms respectively. If the evaluation frequency equals to 2000, then we perform the evaluation of the previous $H = 1000$ points after 2000 points.

Since the drifting is happening in the dataset roughly after 1000 points and after 2000 points the new data distribution is stable (cf. Figure 5.3), the clustering quality is the worst in the part [1000, 2000] from the dataset. This is reflected by most of the measures in both figures. Additionally, it is worthy mentioning that the generally lower values the clustering quality in Figure 10.11 when compared to those of Figure 10.12 is mainly due to the known advantages of PROCLUS when compared with P3C.

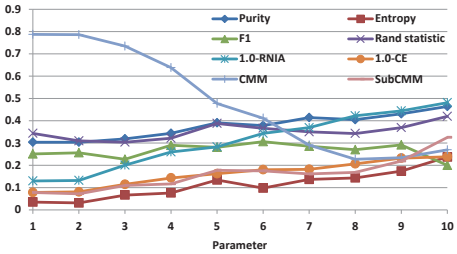


Figure 10.10: Evaluating the DenStream+PROCLUS combination algorithm with different full-space and subspace evaluation measures against SubCMM using a 12-D RBF subspace stream dataset, when varying the number of relevant dimensions (Parameter= number of relevant dimensions).

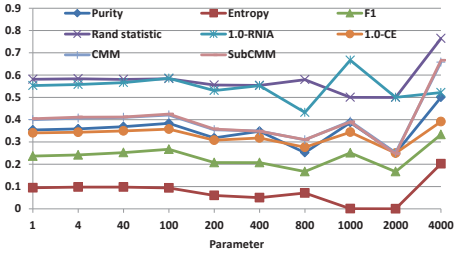


Figure 10.11: Evaluating the DenStream+P3C combination algorithm with different full-space and subspace evaluation measures against SubCMM using the $SynStream3D$ dataset, when varying the evaluation frequency (Parameter= the number of points after which an evaluation is performed).

Figure 10.13 depicts the clustering quality of the PreDeConStream when using the SubCMM and other evaluation measures over the $SynStream3D$ dataset for different evaluation frequencies. Here we notice that generally better clustering quality are achieved by the PreDeConStream than the other algorithm combinations. Again SubCMM is reaching a minimum as most of the other measures when the evaluation frequency is 2000. Figure 10.14 shows the same previous experiment but when varying the stream speed. When the stream speed is too high, PreDeConStream fails to follow the stream changes, and when compared to the ground truth, its clusters look considerably deviating. This is reflected by the lower values of SubCMM for higher stream speeds, and the 0 value of SubCMM when the stream speed is 4000.

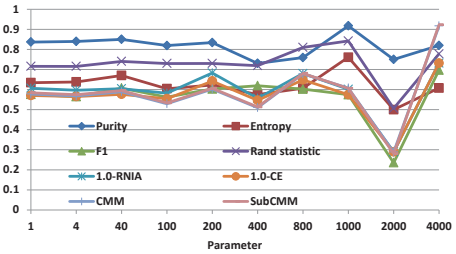


Figure 10.12: Evaluating the DenStream+PROCLUS combination algorithm with different full-space and subspace evaluation measures against SubCMM using the $SynStream3D$ dataset, when varying the evaluation frequency (Parameter= the number of points after which an evaluation is performed).

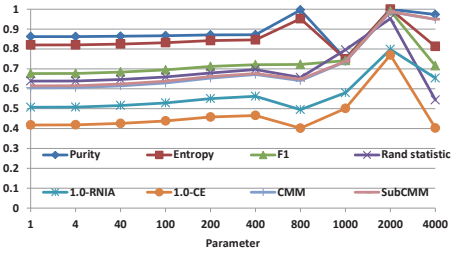


Figure 10.13: Evaluating the PreDeConStream algorithm with different full-space and subspace evaluation measures against SubCMM using the $SynStream3D$ dataset, when varying the evaluation frequency (Parameter= the number of points after which an evaluation is performed).

10.5 Conclusion

In this chapter we have suggested a new way for evaluating stream subspace clustering algorithms by making use of available offline subspace clustering algorithms as well as using the streaming environment to be able to handle streams. Additionally, we have suggested a first subspace clustering measure mainly designed for streaming algorithms. We have thoroughly tested these measures by comparing them to full-space ones. We could show the superiority of most of the suggested measures in the subspace streaming cases.

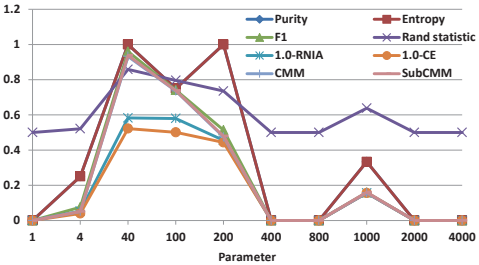


Figure 10.14: Evaluating the PreDeConStream algorithm with different full-space and subspace evaluation measures against SubCMM using the $SynStream3D$ dataset, when varying the stream speed (Parameter= the number of points per time unit).

Part V

Summary

Chapter 11

Summary and Future Work

In this thesis, paradigms and advanced models for stream clustering, anytime mining and subspace clustering were combined and presented. In this chapter, we summarize the contributions given in the different chapters and we give an outlook of the promising future work that can be built above the contributions of this thesis.

11.1 Summary

In the first part of the thesis, we developed three novel methods for an energy-efficient in-sensor-network aggregation of data. In Chapter 2, we presented our novel energy-efficient k-center clustering solution as a single-pass incremental processing algorithm which is aware of outliers. We enhanced the clustering quality by excluding these outlying objects from the clustering. Furthermore, we reduced the cost of intensive reclustering operations and achieved lower energy consumption. For the limited energy resources of sensor networks, our energy-efficient computation induces longer lifetimes of the network. In thorough experiments we presented the high clustering accuracy and low energy consumption of our approach. Furthermore, our algorithm is also aware of limited memory resources in recent sensor nodes. In Chapter 3, we have presented a novel weighted k-center clustering alternative to EDISKCO, called *SenClu*. It is a single-pass algorithm that immediately detects new trends in the drifting sensor data stream and follows them. The light-weighted decaying technique which we used to enhance the clustering quality gives lower influence to old data. As sensor data are usually noisy, *SenClu* is also outlier-aware. In thorough experiments on drifting

synthetic and real world datasets, we showed that *SenClu* outperforms state-of-the-art algorithms by producing higher clustering quality and following trends in the stream, while consuming nearly the same amount of energy. In Chapter 4, we proposed a novel algorithm for an energy-aware physical clustering of sensor nodes. The algorithm considers both spatial and data similarities when building these physical clusters. Nodes in our suggested approach make use of established data mining techniques like subspace clustering for joining physical clusters according to relevant attributes, and outlier detection for online exclusion of outlying readings. We further suggested a powerful self-maintenance method of the constructed clusters. This enables the network to adapt with different changes of observed phenomena in an unsupervised way, while consuming less energy. We proved the efficiency and effectiveness of our approach through comprehensive experiments.

In the second part, we developed density-based stream clustering approaches that tackled the high-dimensionality and the evolving noisy nature of streaming data. In Chapter 5, we have introduced a novel projected stream clustering algorithm termed *PreDeConStream*. Our technique builds a microcluster-based structure to store an online summary of the streaming data. The bottleneck of stream clustering algorithms is usually the offline phase. This is even more critical when having a projected clustering approach applied in the offline phase. Therefore, our algorithm applies an efficient projected clustering by localizing the changes that happened since the previous offline clustering result, and by introducing a novel clustering validity interval. As a result, our technique has proved experimentally its superiority over state-of-the-art techniques. In Chapter 6, we proposed *HASTREAM*, a novel algorithm for hierarchical density-based clustering on evolving data streams. The presented algorithm is able to detect clusters of different densities by adapting the density threshold for each cluster using hierarchical clustering techniques. HASTREAM uses the microcluster structure to represent the data stream in a compact and storage-friendly way. In the offline phase, a hierarchical density-based model is efficiently applied in a streaming fashion by introducing a weighted stability measure of the final flat clustering. To reduce the heavy computational cost in the offline phase of the algorithm, our model additionally applies incremental techniques to update the minimal spanning tree of the graph representing the microclusters. The minimal spanning tree is the basis to extract the hierarchical clustering. The minimal spanning tree is based on the mutual reachability distance. The previously com-

puted core-distances and the distances between some microclusters might still be valid and the recomputation can be avoided. Thus, maintaining the minimal spanning tree reduces the computational costs of the offline phase. Clusters with different sizes and densities are found using an adaptive density threshold that is automatically determined for each cluster by maximizing the cluster stability value. The extensive experimental evaluation study on synthetic and real world datasets showed that HASTREAM is able to find clusters of different densities, evolving nature, and different shapes and sizes. The state-of-the-art competitor was unable to detect such extremely-evolving clusters. The efficiency and the effectiveness experiments showed the superiority of HASTREAM over the state-of-the-art.

In the third part, advanced anytime stream clustering approaches that consider highly-drifting, noisy high-dimensional streaming data were contributed. In Chapter 7, we detailed a novel algorithm for anytime stream clustering called *LiarTree*, which automatically adapts its model size to the stream speed in logarithmic time. It consists of a tree structure that represents detailed information in its leaf nodes and coarser summaries in its inner nodes. The LiarTree avoids overlapping through a local look ahead technique and a reorganization method. It incorporates explicit noise handling on all levels of the hierarchy. It allows the transition from local noise buffers to new entries (i.e., the microclusters) and grows novel subtrees in a top-down manner using its liar concept. This concept makes it robust against noise and changes in the distribution of the underlying stream, and thus, suitable for streaming sensor data clustering. Moreover, the LiarTree as an anytime clustering algorithm, constitutes an anytime algorithm and automatically adapts its model size to the stream speed. In experimental evaluation we have shown on synthetic and real sensor data for various data stream scenarios that the LiarTree outperforms competing approaches in the presence of noise and evolving data, proving its novel concepts to be effective. In Chapter 8, we proposed a novel algorithm for subspace anytime stream clustering called *SubClusTree*. It processes the data objects in a single pass and maintains the microclusters in their relevant subspaces efficiently. It uses a tree structure with a bit-vector to represent a subspace, where the bit-vector keeps track of the dimensions corresponding to the subspace. The tree structure stores finer summaries of data in its leaf nodes and coarser information in its inner nodes. Furthermore, it makes no assumption regarding the number of clusters and can keep a larger number of microclusters. To capture the evolving behavior of the streaming data, it incorpo-

rates a decaying mechanism that allows old concepts to expire. SubClusTree uses a smart strategy to bypass the slow, sequential, bottom-up detection of subspace candidates, by directly jumping to higher, promising subspaces. It takes the advantage of flexible grids and joins them randomly to form subspace candidates, then it prunes the candidates that do not have enough potential to become a cluster. In an extensive experimental evaluation using synthetic and real datasets, we have shown that our approach is rigid to the introduced parameters and has a quite stable clustering quality and a good scalability. It can detect the instant changes in stream and updates the relevant subspaces accordingly. SubClusTree is compatible with the anytime concept, when the time allowances between the readings change.

In the fourth part, we contributed to the area of evaluating stream subspace clustering with a first open-source evaluation framework and a first evaluation measure. In Chapter 9, we introduced the *Subspace MOA* framework as a first open source framework used for embedding, evaluating and visualizing the output of subspace stream clustering algorithms. As there exist only two standalone projected stream clustering algorithms, we have used a special online-reconstruction-offline model that gives the user the possibility of combining their own algorithm out of 3 online summarization methods and 5 offline subspace clustering techniques. In addition to enabling the reading of external datasets, the framework is supported with a subspace stream generator. Multiple state-of-the-art subspace and stream clustering evaluation measures are also included. The framework offers researchers the possibilities to detect pros and contras of different subspace clustering algorithms when applied in the streaming scenario. Additionally, the suitable online-offline combination for a certain dataset can be decided. This is all done in a user friendly interface that is in line with the MOA framework style. In Chapter 10, we contributed a novel external evaluation measure for stream subspace clustering algorithms called *SubCMM*: Subspace Cluster Mapping Measure. SubCMM is able to handle errors caused by emerging, moving, or splitting subspace clusters. Additionally, we extensively compared in this chapter our new measure against state-of-the-art full-space stream clustering evaluation measures. The experimental evaluation, that is performed using the Subspace MOA framework, depicted the ability of *SubCMM* to reflect different changes happening in the subspaces of the evolving stream.

11.2 Future Work

Many further interesting work can be built over the scientific contribution of this thesis. While various paradigms for efficient clustering of high-dimensional streaming data and for anytime stream clustering were introduced in this research, further promising research directions can be established by combining both of these paradigms.

One promising research direction is extending the anytime concept to the offline part of stream clustering algorithms. The offline phase of these algorithms is the bottleneck to delivering the final clustering results. The aspects of localizing the change and updating the previous solutions, introduced in Part II, can be applied to gain an anytime offline phase. This is strongly motivated by scenarios where users are interested of answers within varying allowances of delivery durations.

Another interesting research direction would be allowing the anytime interruption even during the insertion within the sub trees of SubClusTree. Questions about the complexity of the hitchhiking processes and the possibility of jumping between internal levels of the sub trees must be, among others, deeply investigated. Additionally, a projected model of an anytime stream clustering looks a promising combination of the two aspects mentioned in Parts II and III.

Bringing the aspects of data stream processing to graphs is an emerging research topic. In HASTREAM, a novel approach on updating the minimal spanning tree by localizing the changes resulted from inserting and deleting a vertex from the graph, was introduced. This contribution might form a basis for investigating and contributing efficient mining algorithms over evolving attributed graphs.

A further research direction is benefiting from the online-offline stream clustering aspects for dealing with the excessive sizes of big data. One fundamental research question in this context is how to reduce the effect of forcing a certain order of the non-ordered dataset while processing it in a streaming manner. Running carefully-selected sequences of the data in parallel, and efficiently combining the resulted clusterings could be one promising first way of answering the above question.

All streaming objects considered in this research are connected to a single timestamp. One interesting research direction is to consider interval-based objects while processing them using streaming approaches. Each object will be coupled with a starting and an ending timestamp. One promising example is ap-

plying the aspects of streaming sequential pattern mining [AH14] over interval-based objects. Within this context, a natural successive requirement would be offering a streaming sequential pattern mining framework similar to Subspace MOA. The framework must be supported with an interval-based streaming generator, an internal evaluation measure and a suitable visualization interface.

Part VI

Appendices

Bibliography

[ABKS99] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. OPTICS: Ordering points to identify the clustering structure. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 49–60. ACM, 1999.

[ADGK07] Benjamin Arai, Gautam Das, Dimitrios Gunopulos, and Nick Koudas. Anytime measures for top-k algorithms. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 914–925. VLDB Endowment, 2007.

[Agg07] Charu C. Aggarwal. *Data Streams: Models and Algorithms*. Advances in Database Systems. Springer, 2007.

[Agg13] Charu C. Aggarwal. *Managing and Mining Sensor Data*. Springer Publishing Company, Incorporated, 2013.

[AGGR98] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 94–105. ACM, 1998.

[AH14] Charu C. Aggarwal and Jiawei Han. *Frequent Pattern Mining*. Springer International Publishing, 2014.

[AHWY03] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for clustering evolving data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases*, VLDB '03, pages 81–92. VLDB Endowment, 2003.

[AHWY04] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for projected clustering of high dimensional data streams. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 852–863. VLDB Endowment, 2004.

[AKMS07] Ira Assent, Ralph Krieger, Emmanuel Müller, and Thomas Seidl. DUSC: Dimensionality unbiased subspace clustering. In *Proceedings of the 7th IEEE International Conference on Data Mining*, ICDM '07, pages 409–414. IEEE, 2007.

[AKMS08] Ira Assent, Ralph Krieger, Emmanuel Müller, and Thomas Seidl. INSCY: Indexing subspace clusters with in-process-removal of redundancy. In *Proceedings of the 8th IEEE International Conference on Data Mining*, ICDM '08, pages 719–724. IEEE, 2008.

[AW10] Charu C. Aggarwal and Haixun Wang. *Managing and Mining Graph Data*. Advances in Database Systems. Springer, 2010.

[AWY+99] Charu C. Aggarwal, Joel L. Wolf, Philip S. Yu, Cecilia Procopiuc, and Jong Soo Park. Fast algorithms for projected clustering. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 61–72. ACM, 1999.

[BC06] Elena Baralis and Tania Cerquitelli. Selecting representatives in a sensor network. In *In Proceedings of the SEBD*, SEBD '06, pages 351–360, 2006.

[BGH+97] N. L. Bowers, H. U. Gerber, J. C. Hickman, D. A. Jones, and C. J. Nesbitt. *Actuarial Mathematics*. Society of Actuaries, 1997.

[BGRS99] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *Proceedings of the 7th International Conference on Database Theory*, ICDT '99, pages 217–235. Springer-Verlag, 1999.

[BHKP10] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. MOA: Massive online analysis. *J. Mach. Learn. Res.*, 11:1601–1604, 2010.

[BKKK04] Christian Bohm, Karin Kailing, Hans-Peter Kriegel, and Peer Kroger. Density connected clustering with local subspace preferences. In *Proceedings of the 4th IEEE International Conference on Data Mining*, ICDM '04, pages 27–34. IEEE Computer Society, 2004.

[Bor26] Otakar Boruvka. O Jistém Problému Minimálním (About a Certain Minimal Problem) (in Czech, German summary). *Práce Mor. Prírodoved. Spol. v Brne III*, 1926.

[BW08] Lee Byron and Martin Wattenberg. Stacked graphs - geometry & aesthetics. *IEEE Trans. Vis. Comput. Graph.*, 14(6):1245–1252, 2008.

[BZ07] Björn Bringmann and Albrecht Zimmermann. The chosen few: On identifying valuable patterns. In *Proceedings of the 7th IEEE International Conference on Data Mining*, ICDM '07, pages 63–72, 2007.

[CCFM97] Moses Charikar, Chandra Chekuri, Tomás Feder, and Rajeev Motwani. Incremental clustering and dynamic information retrieval. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 626–635. ACM, 1997.

[CEQZ06] Feng Cao, Martin Ester, Weining Qian, and Aoying Zhou. Density-based clustering over an evolving data stream with noise. In *Proceedings of the 6th SIAM International Conference on Data Mining*, SDM '06, pages 328–339, 2006.

[CFZ99] Chun-Hung Cheng, Ada Waichee Fu, and Yi Zhang. Entropy-based subspace clustering for mining numerical data. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '99, pages 84–93. ACM, 1999.

[CH78] Francis Chin and David Houck. Algorithms for updating minimal spanning trees. *Journal of Computer and System Sciences*, 16:333 – 344, 1978.

[Cha00] Bernard Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *J. ACM*, 47(6):1028–1047, 2000.

[CKHS03] Koby Crammer, Jaz Kandola, Royal Holloway, and Yoram Singer. Online classification on a budget. In *Advances in Neural Information Processing Systems 16*. MIT Press, 2003.

[CKMN01] Moses Charikar, Samir Khuller, David M. Mount, and Giri Narasimhan. Algorithms for facility location problems with outliers. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '01, pages 642–651. Society for Industrial and Applied Mathematics, 2001.

[CMS13] Ricardo J. G. B. Campello, Davoud Moulavi, and Jörg Sander. Density-based clustering based on hierarchical density estimates. In *Advances in Knowledge Discovery and Data Mining, 17th Pacific-Asia Conference, Part II*, PAKDD '13, pages 160–172, 2013.

[CMZ07] Graham Cormode, S. Muthukrishnan, and Wei Zhuang. Conquering the divide: Continuous clustering of distributed data streams. In *IEEE 23rd International Conference on Data Engineering*, ICDE '07, pages 1036–1045. IEEE Computer Society, 2007.

[COP03] Moses Charikar, Liadan O'Callaghan, and Rina Panigrahy. Better streaming algorithms for clustering problems. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, STOC '03, pages 30–39. ACM, 2003.

[CT07] Yixin Chen and Li Tu. Density-based clustering for real-time stream data. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 133–142. ACM, 2007.

[Dat99] Network Intrusion Dataset. KDD Cup Data. `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`, 1999.

[Dat04] Intel Dataset. Dataset of Intel Berkeley Research Lab, 2004.

[DeC02] Dennis DeCoste. Anytime interval-valued outputs for kernel machines: Fast support vector machine classification via distance geometry. In *Proceedings of the 19th International Conference on Machine Learning*, ICML '02, pages 99–106, 2002.

[Dij59] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, pages 269–271, 1959.

[DL01] Bevan Das and Michael C. Loui. Reconstructing a minimum spanning tree after deletion of any node. *Algorithmica*, 31(4):530–547, 2001.

[EKSX96] Martin Ester, Hans-Peter Kriegel, Jorg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *The 2nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '96, pages 226–231. AAAI Press, 1996.

[FG88] Tomás Feder and Daniel Greene. Optimal algorithms for approximate clustering. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 434–444. ACM, 1988.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, pages 596–615, 1987.

[GE07] Joao Gama and Mohamed M. Gaber (Eds). *Learning from Data Streams – Processing techniques in Sensor Networks*. Springer, 2007.

[GGO+08] Auroop R. Ganguly, Joao Gama, Olufemi A. Omitaomu, Mohamed Medhat Gaber, and Ranga Raju Vatsavai. *Knowledge Discovery from Sensor Data*. CRC Press, Inc., 1st edition, 2008.

[Gon85] Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38(2-3):293–306, 1985.

[Guh09] Sudipto Guha. Tight results for clustering and summarizing data streams. In *Proceedings of the 12th International Conference on Database Theory*, ICDT '09, pages 268–275. ACM, 2009.

[GZJ06] Oleksandr Grygorash, Yan Zhou, and Zach Jorgensen. Minimum spanning tree based clustering algorithms. In *Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '06, pages 73–81. IEEE Computer Society, 2006.

[HB99] S. Hettich and S. Bay. The UCI KDD archive http://kdd.ics.uci.edu, 1999.

[HCB00] Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *Proceedings of the 33rd Hawaii International Conference on System Sciences*, HICSS '00, pages 3005–3014. IEEE Computer Society, 2000.

[HD03] Geoff Hulten and Pedro Domingos. VFML – a toolkit for mining high-speed time-changing data streams. http://www.cs.washington.edu/dm/vfml/, 2003.

[HFH+09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Exploration Newsletter*, 11(1):10–18, 2009.

[HKCS13] Marwan Hassani, Yunsu Kim, Seungjin Choi, and Thomas Seidl. Effective evaluation measures for subspace clustering of data streams. In *Trends and Applications in Knowledge Discovery and Data Mining - PAKDD 2013 International Workshops*, pages 342–353, 2013.

[HKP06] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques, Second Edition*. Morgan Kaufmann Publishers Inc., 2006.

[HKS11] Marwan Hassani, Philipp Kranen, and Thomas Seidl. Precise anytime clustering of noisy sensor data with logarithmic complexity. In *Proceedings of the 5th International Workshop on Knowledge Discovery from Sensor Data*, SensorKDD '11 @KDD '11, pages 52–60. ACM, 2011.

[HKS13] Marwan Hassani, Yunsu Kim, and Thomas Seidl. *Subspace MOA:* subspace stream clustering evaluation using the MOA framework. In *Database Systems for Advanced Applications, 18th International Conference, DASFAA (2)*, pages 446–449, 2013.

[HKSS14] Marwan Hassani, Philipp Kranen, Rajveer Saini, and Thomas Seidl. Subspace anytime stream clustering. In *Proceedings of the 26th Con-*

ference on Scientific and Statistical Database Management, SSDBM' 14, page 37, 2014.

[HMS09] Marwan Hassani, Emmanuel Müller, and Thomas Seidl. EDISKCO: Energy Efficient Distributed In-Sensor-Network K-center Clustering with Outliers. In *Proceedings of the 3rd International Workshop on Knowledge Discovery from Sensor Data*, SensorKDD '09 @KDD '09, pages 39–48. ACM, 2009.

[HMS+10] Marwan Hassani, Emmanuel Müller, Pascal Spaus, Adriola Faqolli, Themis Palpanas, and Thomas Seidl. Self-organizing energy aware clustering of nodes in sensor networks using relevant attributes. In *Proceedings of the 4th International Workshop on Knowledge Discovery from Sensor Data*, SensorKDD '10 @KDD '10, pages 39–48. ACM, 2010.

[HS85] Dorit S. Hochbaum and David B. Shmoys. A best possible approximation algorithm for the k-centre problem. *Math. of Operations Research*, 10:180–184, 1985.

[HS11] Marwan Hassani and Thomas Seidl. Towards a mobile health context prediction: Sequential pattern mining in multiple streams. In *Proceedings of the IEEE 12th International Conference on Mobile Data Management*, volume 2 of *MDM '11*, pages 55–57. IEEE Computer Society, 2011.

[HS12a] Marwan Hassani and Thomas Seidl. Distributed weighted clustering of evolving sensor data streams with noise. *Journal of Digital Information Management (JDIM)*, 10(6):410–420, 2012.

[HS12b] Marwan Hassani and Thomas Seidl. Resource-aware distributed clustering of drifting sensor data streams. In *Proceedings of the 4th International Conference on Networked Digital Technologies*, NDT' 12, pages 592–607, 2012.

[HS14] Marwan Hassani and Thomas Seidl. Efficient streaming detection of hidden clusters in big data using subspace stream clustering. In *Proceedings of the 19th International Conference on Database Systems for Advanced Applications (DASFAA) - Workshops*, pages 146–160, 2014.

[HSGS12] Marwan Hassani, Pascal Spaus, Mohamed Medhat Gaber, and Thomas Seidl. Density-based projected clustering of data streams. In *Proceedings of the 6th International Conference on Scalable Uncertainty Management*, SUM '12, pages 311–324, 2012.

[HSS14] Marwan Hassani, Pascal Spaus, and Thomas Seidl. Adaptive multiple-resolution stream clustering. In *Proceedings of the 10th International Conference on Machine Learning and Data Mining*, MLDM '14, pages 134–148, 2014.

[HTGS14] Marwan Hassani, Ayman Tarakji, Lyubomir Georgiev, and Thomas Seidl. Parallel implementation of a density-based stream clustering algorithm over a GPU scheduling system. In *Proceedings of the Workshop on Scalable Data Analytics: Theory and Applications SDA'14 @ PAKDD '14*, pages 441–453, 2014.

[JAB98] Charles W. Anderson Jock A. Blackard, Denis J. Dean. UCI machine learning repository, http://archive.ics.uci.edu/ml/datasets/covertype, 1998.

[JGA06] Ruoming Jin, Anjan Goswami, and Gagan Agrawal. Fast and exact out-of-core and distributed k-means clustering. *Knowledge and Information Systems*, 10(1):17–40, July 2006.

[JKP04] Eshref Januzaj, Hans-Peter Kriegel, and Martin Pfeifle. Scalable density-based distributed clustering. In *Proceedings 8th European Conference on Principles and Practice of Knowledge Discovery in Databases*, PKDD '09, pages 231–244, 2004.

[JZC06] Ankur Jain, Zhihua Zhang, and Edward Y. Chang. Adaptive non-linear clustering in data streams. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, CIKM '06, pages 122–131. ACM, 2006.

[KABS09] Philipp Kranen, Ira Assent, Corinna Baldauf, and Thomas Seidl. Self-adaptive anytime stream clustering. In *Proceedings of The 9th IEEE International Conference on Data Mining*, ICDM '09, pages 249–258, 2009.

[KGFS10] Philipp Kranen, Stephan Guñnemann, Sergej Fries, and Thomas Seidl. MC-tree: Improving bayesian anytime classification. In *Proceedings of the 22nd Scientific and Statistical Database Management*, SSDBM '10, pages 252–269. Springer Berlin Heidelberg, 2010.

[KKJ+11] Hardy Kremer, Philipp Kranen, Timm Jansen, Thomas Seidl, Albert Bifet, Geoff Holmes, and Bernhard Pfahringer. An effective evaluation measure for clustering on evolving data streams. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '11, pages 868–876, 2011.

[KKK04] Karin Kailing, Hans-Peter Kriegel, and Peer Kröger. Density-connected subspace clustering for high-dimensional data. In *Proceedings of the SIAM International Conference on Data Mining*, SDM '04, pages 246–257, 2004.

[KKNZ10] Hans-Peter Kriegel, Peer Kröger, Irene Ntoutsi, and Arthur Zimek. Towards subspace clustering on dynamic data: an incremental version of PreDeCon. In *Proc. of 1st StreamKDD workshop in conj. with KDD '10*, StreamKDD '10, pages 31–38. ACM, 2010.

[KKRW05] Hans-Peter Kriegel, Peer Kröger, Matthias Renz, and Sebastian Wurst. A generic framework for efficient subspace clustering of high-dimensional data. In *Proceedings of the 5th IEEE International Conference on Data mining*, ICDM '05, pages 250–257. IEEE Computer Society, 2005.

[KKT95] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, pages 321–328, 1995.

[KKZ09] Hans-Peter Kriegel, Peer Kröger, and Arthur Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *TKDD: ACM Trans. Knowl. Discov. Data*, 3(1):1:1–1:58, 2009.

[Kot05] Yanis Kotidis. Snapshot queries: Towards data-centric sensor networks. In *Proceeding of the 21st International Conference on Data Engineering*, ICDE '05, pages 131 – 142. IEEE Computer Society, 2005.

[KR90] Leonard Kaufman and Peter J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. Wiley series in probability and mathematical statistics: Applied probability and statistics. John Wiley, 1990.

[Kru56] Joseph B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society*, pages 48–50, 1956.

[KS09] Philipp Kranen and Thomas Seidl. Harnessing the strengths of anytime algorithms for constant data streams. In *Data Mining and Knowledge Discovery Journal (DMKD), Special Issue on Selected Papers from ECML PKDD 2009, Vol. 19, No. 2*, pages 245–260, 2009.

[LC08] Guopin Lin and Leisong Chen. A grid and fractal dimension-based data stream clustering algorithm. In *ISISE '08*, pages 66 –70, 2008.

[LXY00] Bing Liu, Yiyuan Xia, and Philip S. Yu. Clustering through decision tree construction. In *Proceedings of the 9th International Conference on Information and Knowledge Management*, CIKM '00, pages 20–29. ACM, 2000.

[MAG+09a] Emmanuel Müller, Ira Assent, Stephan Günnemann, Timm Jansen, and Thomas Seidl. Opensubspace: An open source framework for evaluation and exploration of subspace clustering algorithms in weka. In *In Open Source in Data Mining Workshop at PAKDD*, pages 2–13, 2009.

[MAG+09b] Emmanuel Müller, Ira Assent, Stephan Günnemann, Ralph Krieger, and Thomas Seidl. Relevant subspace clustering: Mining the most interesting non-redundant concepts in high dimensional data. In *IEEE 13th International Conference on Data Mining*, ICDM '09, pages 377–386. IEEE Computer Society, 2009.

[MAK+09] Emmanuel Müller, Ira Assent, Ralph Krieger, Stephan Günnemann, and Thomas Seidl. DensEst: Density estimation for data mining in high dimensional spaces. In *Proceedings of the SIAM International Conference on Data Mining*, SDM '09, pages 175–186, 2009.

[MGAS09] Emmanuel Müller, Stephan Günnemann, Ira Assent, and Thomas Seidl. Evaluating clustering in subspace projections of high dimensional data. *PVLDB*, 2(1):1270–1281, 2009.

[MK08] Richard Matthew McCutchen and Samir Khuller. Streaming algorithms for k-center clustering with outliers and with anonymity. In *APPROX-RANDOM*, volume 5171 of *Lecture Notes in Computer Science*, pages 165–178. Springer, 2008.

[MS91] Dietrich Werner Müller and G. Sawitzki. Excess mass estimates and tests for multimodality. *Journal of the American Statistical Association*, pages 738–746, 1991.

[MS06] Anand Meka and Ambuj K. Singh. Distributed spatial clustering in sensor networks. In *EDBT 2006, LNCS 3896*, volume 3896 of *Lecture Notes in Computer Science*, pages 980–1000. Springer, 2006.

[MSE06] Gabriela Moise, Joerg Sander, and Martin Ester. P3C: A robust projected clustering algorithm. In *Proceedings of the 6th IEEE International Conference on Data Mining*, ICDM '07, pages 414–425. IEEE, 2006.

[NGC01] Harsha Nagesh, Sanjay Goil, and Alok Choudhary. Adaptive grids for clustering massive data sets. In *Proceedings of the First SIAM International Conference on Data Mining*, SDM '01, pages 1–17, 2001.

[NPW04] Enrico Nardelli, Guido Proietti, and Peter Widmayer. Nearly linear time minimum spanning tree maintenance for transient node failures. *Algorithmica*, pages 119–132, 2004.

[NZP+12] Irene Ntoutsi, Arthur Zimek, Themis Palpanas, Peer Kröger, and Hans-Peter Kriegel. Density-based projected clustering over high dimensional data streams. In *Proceedings of the 12th SIAM International Conference on Data Mining*, SDM '12, pages 987–998, 2012.

[OY04] Sonia Fahmy Ossama Younis. Heed: A hybrid, energy-efficient, distributed clustering approach for ad hoc sensor networks. *IEEE Transactions on Mobile Computing*, 3(4):366–379, 10 2004.

[PHL04] Lance Parsons, Ehtesham Haque, and Huan Liu. Subspace clustering for high dimensional data: a review. *SIGKDD Explorations*, 6(1):90–105, 2004.

[Phy] Physiological. Dataset. `http://www.cs.purdue.edu/commugrate/data/2004icml/`.

[PL07] Nam Hun Park and Won Suk Lee. Grid-based subspace clustering over data streams. In *Proceedings of the 16th ACM Conference on Information and Knowledge Management*, CIKM '07, pages 801–810, 2007.

[PLL07] Dragoljub Pokrajac, Aleksandar Lazarevic, and Longin Jan Latecki. Incremental local outlier detection for data streams. In *Proceedings of the IEEE symposium on Computational Intelligence and Data Mining*, CIDM '07, pages 504–515, 2007.

[PM06] Anne Patrikainen and Marina Meila. Comparing subspace clusterings. *IEEE Transactions on Knowledge and Data Engineering*, 18(7):902–916, 2006.

[Pri57] Robert C. Prim. Shortest connection networks and some generalizations. *The Bell Systems Technical Journal*, pages 1389–1401, 1957.

[PSC05] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: Enabling ultra-low power wireless research. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, IPSN '05. IEEE Press, 2005.

[QBG+13] Christoph Quix, Johannes Barnickel, Sandra Geisler, Marwan Hassani, Saim Kim, Xiang Li, Andreas Lorenz, Till Quadflieg, Thomas Gries, Matthias Jarke, Steffen Leonhardt, Ulrike Meyer, and Thomas Seidl. Healthnet: A system for mobile and wearable health information management. In *Proceedings of the 3rd International Workshop on Information Management for Mobile Applications*, pages 36–43, 2013.

[RGL08] Pedro Pereira Rodrigues, João Gama, and Luís M. B. Lopes. Clustering distributed sensor data streams. In *Machine Learning and*

Knowledge Discovery in Databases, European Conference, ECML PKDD '08, pages 282–297, 2008.

[SAK+09] Thomas Seidl, Ira Assent, Philipp Kranen, Ralph Krieger, and Jennifer Herrmann. Indexing density models for incremental learning and anytime classification on data streams. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 311–322. ACM, 2009.

[SBF+07] Adam Silberstein, Rebecca Braynard, Gregory Filpus, Gavino Puggioni, Alan Gelfand, Kamesh Munagala, and Jun Yang. Data-driven processing in sensor networks. In *Proceedings of the 3rd Biennal Conference on Innovative Data Systems Research*, CIDR '07, pages 588–599, 2007.

[SK01] W. Nick Street and Yong Seog Kim. A streaming ensemble algorithm (sea) for large-scale classification. In *Proceedings of the 7th ACM Conference on Knowledge Discovery and Data mining*, KDD '01, pages 377–382, 2001.

[SK10] Jin Shieh and Eamonn Keogh. Polishing the right apple: Anytime classification also benefits data streams with constant arrival times. In *Proceedings of the 10th IEEE International Conference on Data Mining*, ICDM '10, pages 461–470, 2010.

[SZ05] Karlton Sequeira and Mohammed Zaki. SCHISM: A new approach to interesting subspace mining. *International Journal of Business Intelligence and Data Mining*, 1(2):137–160, 2005.

[TGC06] Andrew S. Tanenbaum, Chandana Gamage, and Bruno Crispo. Taking sensor networks from the lab to the jungle. *IEEE Computer*, 39(8):98–100, 2006.

[TRA07] Dimitris K. Tasoulis, Gordon Ross, and Niall M. Adams. Visualising the cluster structure of data streams. In *Advances in Intelligent Data Analysis VII*, pages 81–92, 2007.

[UXKL06] Ken Ueno, Xiaopeng Xi, Eamonn J. Keogh, and Dah-Jye Lee. Anytime classification using the nearest neighbor algorithm with ap-

plications to stream mining. In *Proceedings of the 6th IEEE International Conference on Data Mining*, ICDM '06, pages 623–632, 2006.

[VNT+14] Tobias Vaegs, Paula Niemietz, Christian Tummel, Anja Richert, Christian Beecks, Max Haberstroh, Marwan Hassani, Tobias Meisen, Matthias Priesters, Helmut Vieritz, Thomas Seidl, Irene Mittelberg, Stella Neumann, and Sabina Jeschke. Fostering interdisciplinary integration in the e-humanities. In *Proceedings of the 6th International Conference on Education and New Learning Technologies*, EDULEARN '14, pages 2244–2251. IATED, 2014.

[WFYH03] Haixun Wang, Wei Fan, Philip S. Yu, and Jiawei Han. Mining concept-drifting data streams using ensemble classifiers. In *Proceedings of the 9th ACM Conference on Knowledge Discovery and Data mining*, KDD '03, pages 226–235, 2003.

[WND+09] Li Wan, Wee Keong Ng, Xuan Hong Dang, Philip S. Yu, and Kuan Zhang. Density-based clustering of data streams at multiple resolutions. *ACM Transactions on Knowledge Discovery from Data*, 3(3):14:1–14:28, 2009.

[WWW12] Xiaochun Wang, Xiali Wang, and D. Mitch Wilkes. A minimum spanning tree-inspired clustering-based outlier detection technique. In *Proceedings of the 12th IEEE International Conference on Data Mining*, ICDM '12, pages 209–223. Springer, 2012.

[YG08] Jie Yin and Mohamed Medhat Gaber. Clustering distributed time series in sensor networks. In *Proceedings of the 8th IEEE Conference on Data Mining*, ICDM '08, pages 678–687, 2008.

[YWKMN09] Lexiang Ye, Xiaoyue Wang, Eamonn J. Keogh, and Agenor Mafra-Neto. Autocannibalistic and anyspace indexing algorithms with application to sensor data mining. In *Proceedings of the 9th SIAM International Conference on Data Mining*, SDM '09, pages 85–96, 2009.

[YWKT07] Ying Yang, Geoffrey I. Webb, Kevin B. Korb, and Kai Ming Ting. Classifying under computational resource constraints: anytime classification using probabilistic estimators. *Machine Learning*, 69(1):35–53, 2007.

[ZJ14] Mohammed J. Zaki and Wagner Meira Jr. *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge University Press, New York, NY, USA, 2014.

[ZK04] Ying Zhao and George Karypis. Empirical and theoretical comparisons of selected criterion functions for document clustering. *Machine Learning*, 55(3):311–331, 2004.

[ZRL96] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: An efficient data clustering method for very large databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 103–114. ACM, 1996.

Statement of Originality

This thesis would not have been possible without the close collaboration within the group of Professor Seidl. Many of the presented ideas and techniques evolved from the fruitful discussion in the group. The high level of productive collaboration within the group and also with the students makes it hard to pinpoint the individual contribution. The following provides some more detail on collaboration and support for the individual chapters.

The approaches in Chapters 2 were developed under a great support from Emmanuel Müller. The ECLUN algorithm in Chapter 4 was analyzed and preliminarily implemented in the master thesis of Adriola Faqolli, whom I advised together with Emmanuel Müller and Professor Themis Palpanas from the University of Trento, Italy. The experimental part was extended with a great support from Pascal Spaus. Many approaches in Chapters 5 and 6 were investigated and initially implemented, respectively, in the bachelor and in the master thesis of Pascal Spaus. Ideas of Chapter 5 were discussed with Assoc. Professor Mohamed Gaber during my research stay at the Portsmouth University, UK. Chapters 7 and 8 were developed together with Philipp Kranen who advised with me Rajveer Saini in her diploma thesis on the SubClusTree which was further discussed in Chapter 8. A great support in the analysis and the implementation of Chapter 7 was given by Sanchez Villaamil and Felix Reidl. The Subspace MOA framework in Chapter 9 was implemented with a strong support from Yunsu Kim. The SubCMM measure in Chapter 10 was initially discussed in the bachelor thesis of Yunsu Kim, whom I advised together with Professor Seungjin Choi from the POSTECH University, Korea.

The publications, where the chapters or parts of them appeared, are footnoted at the beginning of the corresponding chapter. A complete list of my previous publications is given separately in the following for convenience.

List of Publications

[AHBS14] Roland Assam, Marwan Hassani, Michael Brysch and Thomas Seidl. (k, d)-Core Anonymity: Structural Anonymization of Massive Networks. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management, (SSDBM '14)*, Article 17, 2014.

[AHS12] Roland Assam, Marwan Hassani and Thomas Seidl. Differential Private Trajectory Protection of Moving Objects. In *Proceedings of the 3rd International Workshop on GeoStreaming, (IWGS '12 @SIGSPATIAL '12)*, pages 68–77, 2012.

[AHS12a] Roland Assam, Marwan Hassani and Thomas Seidl. Differential Private Trajectory Obfuscation. In *Proceedings of the 9th International Conference on Mobile and Ubiquitous Systems, (MobiQuitous '12)*, pages 139–151, 2012.

[HBT+15] Marwan Hassani, Christian Beecks, Daniel Töws, Tatiana Serbina, Max Haberstroh, Paula Niemietz, Sabina Jeschke, Stella Neumann, and Thomas Seidl. Sequential Pattern Mining of Multimodal Streams in the Humanities. In *Proceedings of the 16th Conference on Database Systems for Business, Technology, and Web, (BTW '15)*, pages 683–686, 2015.

[HKCS13] Marwan Hassani, Yunsu Kim, Seungjin Choi, and Thomas Seidl. Effective Evaluation Measures for Subspace Clustering of Data Streams. In *Trends and Applications in Knowledge Discovery and Data Mining - (PAKDD '13) International Workshops*, pages 342–353, 2013.

[HKCS14] Marwan Hassani, Yunsu Kim, Seungjin Choi, and Thomas Seidl. Subspace Clustering of Data Streams: New Algorithms and Effective Evaluation Measures. In *Journal of Intelligent Information Systems, (JIIS)*, 42:1-17, 08 June 2014.

[HKS11] Marwan Hassani, Philipp Kranen, and Thomas Seidl. Precise Anytime Clustering of Noisy Sensor Data With Logarithmic Complexity. In *Proceedings of the 5th International Workshop on Knowledge Discovery from Sensor Data, (SensorKDD '11 @KDD'11)*, pages 52–60. ACM, 2011.

[HKS11a] Marwan Hassani, Philipp Kranen, and Thomas Seidl. Noise-Aware Concise Clustering of Streaming Sensor Data in a Logarithmic Time. In *Proceedings of the 13th Workshop on Knowledge Discovery, Data Mining and Machine Learning, (KDML '11 @LWA '11)*, pages 97–105, 2011.

[HKS13] Marwan Hassani, Yunsu Kim, and Thomas Seidl. Subspace MOA: Subspace Stream Clustering Evaluation Using the MOA Framework. *Database Systems for Advanced Applications, 18th International Conference, (DASFAA '13)*, pages 446–449, 2013.

[HKSS14] Marwan Hassani, Philipp Kranen, Rajveer Saini, and Thomas Seidl. Subspace Anytime Stream Clustering. In *Proceedings of the 26th Conference on Scientific and Statistical Database Management, (SSDBM '14)*, Article 37, 2014.

[HMS09] Marwan Hassani, Emmanuel Müller, and Thomas Seidl. EDISKCO: Energy Efficient Distributed In-Sensor-Network K-center Clustering with Outliers. In *Proceedings of the 3rd International Workshop on Knowledge Discovery from Sensor Data, (SensorKDD '09 @KDD '09)*, pages 39–48. ACM, 2009.

[HMS+10] Marwan Hassani, Emmanuel Müller, Pascal Spaus, Adriola Faqolli, Themis Palpanas, and Thomas Seidl. Self-Organizing Energy Aware Clustering of Nodes in Sensor Networks Using Relevant Attributes. In *Proceedings of the 4th International Workshop on Knowledge Discovery from Sensor Data, (SensorKDD '10 @KDD '10)*, pages 39–48. ACM, 2010.

[HS11] Marwan Hassani and Thomas Seidl. Towards a Mobile Health

Context Prediction: Sequential Pattern Mining in Multiple Streams. In *Proceedings of the IEEE 12th International Conference on Mobile Data Management (2), (MDM '11)*, pages 55–57. IEEE Computer Society, 2011.

[HS12a] Marwan Hassani and Thomas Seidl. Distributed Weighted Clustering of Evolving Sensor Data Streams With Noise. In *Journal of Digital Information Management, (JDIM)*, 10(6):410–420, 2012.

[HS12b] Marwan Hassani and Thomas Seidl. Resource-Aware Distributed Clustering of Drifting Sensor Data Streams. In *Proceedings of the 4th International Conference on Networked Digital Technologies, (NDT '13)*, pages 592–607, 2012.

[HS14] Marwan Hassani and Thomas Seidl. Efficient Streaming Detection of Hidden Clusters in Big Data Using Subspace Stream Clustering. In *Proceedings of the 19th International Conference on Database Systems for Advanced Applications, (DASFAA) - Workshops*, pages 146–160, 2014.

[HS15] Marwan Hassani and Thomas Seidl. Internal Clustering Evaluation of Data Streams. In *Trends and Applications in Knowledge Discovery and Data Mining - (PAKDD '15) International Workshops*, to appear, 2015.

[HSGS12] Marwan Hassani, Pascal Spaus, Mohamed Medhat Gaber, and Thomas Seidl. Density-Based Projected Clustering of Data Streams. In *Proceedings of the 6th International Conference on Scalable Uncertainty Management, (SUM '12)*, pages 311–324, 2012.

[HSS14] Marwan Hassani, Pascal Spaus, and Thomas Seidl. Adaptive Multiple-Resolution Stream Clustering. In *Proceedings of the 10th International Conference on Machine Learning and Data Mining, (MLDM '14)*, pages 134–148, 2014.

[HTGS14] Marwan Hassani, Ayman Tarakji, Lyubomir Georgiev and Thomas Seidl. Parallel Implementation of a Density-Based Stream Clustering Algorithm over a GPU Scheduling System.

In Proceedings of the Workshop on Scalable Data Analytics: Theory and Applications, (SDA '14 @ PAKDD '14), pages 441–453, 2014.

[KHS12] Philipp Kranen, Marwan Hassani and Thomas Seidl. BT* - An Advanced Algorithm for Anytime Classification. In *Proceedings of the 24th International Conference on Scientific and Statistical Database Management, (SSDBM '12)*, pages 289–315, 2012.

[QBG+13] Christoph Quix, Johannes Barnickel, Sandra Geisler, Marwan Hassani, Saim Kim, Xiang Li, Andreas Lorenz, Till Quadflieg, Thomas Gries, Matthias Jarke, Steffen Leonhardt, Ulrike Meyer, and Thomas Seidl. Healthnet: A System for Mobile and Wearable Health Information Management. In *Proceedings of the 3rd International Workshop on Information Management for Mobile Applications, (IMMoA '13)*, pages 36–43, 2013.

[SHB14] Thomas Seidl, Marwan Hassani, and Christian Beecks, editors. Proceedings of the 16th LWA Workshops: KDML, IR and FGWM, Aachen, Germany, September 8-10, 2014. *Volume 1226 of CEUR Workshop Proceedings*, 2014.

[THBS15] Daniel Töws, Marwan Hassani, Christian Beecks and Thomas Seidl. Optimizing Sequential Pattern Mining Within Multiple Streams. In *Proceedings of the Student Program of the 16th Conference on Database Systems for Business, Technology, and Web, (BTW '15)*, to appear, 2015.

[THGS15] Ayman Tarakji, Marwan Hassani, Lyubomir Georgiev and Thomas Seidl. Parallel Density-Based Stream Clustering Using a Multi-User GPU Scheduler. In *Proceedings of the 11th International Conference: Beyond Databases, Architectures and Structures, (BDAS '15)*, pages 343–360, 2015.

[THLS13] Ayman Tarakji, Marwan Hassani, Stefan Lankes and Thomas Seidl. Using a Multitasking GPU Environment for Content-Based Similarity Measures of Big Data. In *Proceedings of the 13th International Conference on Computational Science and its Applications, (ICCSA '13)*, pages 181–196, 2013.

[VNT+14] Tobias Vaegs, Paula Niemietz, Christian Tummel, Anja Richert,

Christian Beecks, Max Haberstroh, Marwan Hassani, Tobias Meisen, Matthias Priesters, Helmut Vieritz, Thomas Seidl, Irene Mittelberg, Stella Neumann and Sabina Jeschke. Fostering Interdisciplinary Integration in the E-Humanities. In *Proceedings of the 6th International Conference on Education and New Learning Technologies, (EDULEARN '14)*, pages 2244–2251, 2014.