FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Dynamic Log File Analysis: An Unsupervised Cluster Evolution Approach for Anomaly Detection

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Business Informatics

eingereicht von

Max Landauer, BSc
Matrikelnummer 01228830

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.-Prof. Dipl.-Ing. Dr.techn. Peter Filzmoser
Mitwirkung: Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Dr.rer.soc.oec. Florian Skopik
Dipl.-Ing. Markus Wurzenberger

Wien, 12. Februar 2018

Max Landauer

Peter Filzmoser

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Dynamic Log File Analysis: An Unsupervised Cluster Evolution Approach for Anomaly Detection

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Max Landauer, BSc
Registration Number 01228830

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Univ.-Prof. Dipl.-Ing. Dr.techn. Peter Filzmoser
Assistance: Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Dr.rer.soc.oec. Florian Skopik
Dipl.-Ing. Markus Wurzenberger

Vienna, 12th February, 2018

Max Landauer

Peter Filzmoser

Erklärung zur Verfassung der Arbeit

Max Landauer, BSc
Hauptplatz 5, 7503 Großpetersdorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 12. Februar 2018

Max Landauer

Acknowledgements

I am very grateful that I had the possibility to write this thesis in cooperation with the Austrian Institute of Technology (AIT). In particular I would like to thank Florian Skopik for giving me the opportunity to carry out the research over the course of an internship and providing me with the necessary insights and expertise for completing this project. Moreover, I would like to sincerely thank Markus Wurzenberger for his continuous advice and without whose support and ideas this thesis could not have been written. I also thank Giuseppe Settanni for helping me setting up the evaluation environment. I would like to thank all my colleagues at AIT for the excellent collaboration in the last months.

I would like to thank my academic advisor Prof. Peter Filzmoser for his helpful ideas and comments.

Furthermore, I thank my aunt Anneliese Steiner for proof-reading this thesis.

Last but not least I would like to sincerely thank my family and friends for always supporting me throughout my life.

Kurzfassung

Technologische Fortschritte und die zunehmende Vernetzung von Computersystemen haben zu einer erhöhten Gefahr durch vormals unbekannte Bedrohungen und Eindringungen über komplexe Angriffsvektoren geführt. Im Bereich von Cyber Security werden aus diesem Grund Intrusion Detection Systems zur Echtzeit-Überwachung von kontinuierlich generierten Logzeilen verwendet um Systeme vor Angriffen zu schützen. Solche existierenden Ansätze verwenden Clustering-Methoden die auf String-Metriken basieren um ähnliche Logzeilen ohne die Notwendigkeit von Parsern zu gruppieren. Dabei werden ungewöhnliche Logzeilen unabhängig von der zugrundeliegenden Syntax oder Semantik der Logdatei als Ausreißer erkannt. Diese Ansätze erzeugen jedoch nur eine statische Sicht auf die Daten und berücksichtigen die dynamische Natur von Protokollzeilen nicht ausreichend. Änderungen in der Systemumgebung oder der technologischen Infrastruktur erfordern daher häufig eine Neuformung der bestehenden Gruppen. Darüber hinaus sind solche Ansätze nicht für die Erkennung von Anomalien bezüglich der Frequenz, Änderungen des periodischen Verhaltens oder Abhängigkeiten von Logzeilen geeignet.

Um diesen Problemen entgegenzuwirken wird in dieser Arbeit eine Methode zur Erkennung von dynamischen Anomalien in Logdateien vorgestellt. Das Verfahren gruppiert ähnliche Logzeilen innerhalb vordefinierter Zeitfenster unter Verwendung eines inkrementellen Clustering-Algorithmus. Dabei werden durch den neuartigen Clustering-Mechanismus Verbindungen zwischen den ansonsten isolierten Ansammlungen von Gruppen hergestellt. Diese Verbindungen zwischen zwei benachbarten Zeitfenstern werden unter Zuhilfenahme von Cluster-Evolutionstechniken analysiert um Übergänge, wie etwa Teilungen oder Fusionen, zu bestimmen. Ein selbstlernender Algorithmus erkennt anschließend Anomalien im zeitlichen Verhalten dieser evolutionären Gruppen indem Metriken aus deren Entwicklungen abgeleitet und analysiert werden.

Ein Prototyp für die oben genannte Methodik wurde im Rahmen dieser Arbeit entwickelt und anhand einer Logdatei mit bekannten Anomalien in einem illustrativen Szenario angewandt. Die Ergebnisse der Evaluierung wurden bezüglich der Einflüsse bestimmter Parameter auf die Anomalieerkennungsfähigkeit sowie die Laufzeit analysiert. Die Evaluierung des Szenarios zeigte, dass die Methodik 61.8% der dynamischen Änderungen der Logzeilen-Cluster korrekt identifizieren konnte, wobei die Fehlalarmrate nur 0.7% betrug. Ein effizientes Erkennen solcher Anomalien und die Fähigkeit der Selbstanpassung bei technologischen Änderungen begründen die Anwendbarkeit des vorgestellten Ansatzes.

Abstract

Technological advances and the increased interconnectivity of computer systems have led to a higher risk of previously unknown threats and intrusions through diverse attack vectors. Cyber security therefore employs Intrusion Detection Systems that monitor continuously generated log lines in real-time in order to protect systems from such attacks. Existing approaches use clustering techniques based on string metrics in order to group similar log lines into clusters without any need for parsers. Thereby, dissimilar lines are detected as outliers independent from the syntax and semantics of the log file. However, such methods only produce a static view on the data and do not sufficiently incorporate the dynamic nature of computer logs. Changes of the system environment or technological infrastructure therefore frequently require cluster reformations. Moreover, such approaches are not suited for detecting anomalies related to frequencies, periodic alterations and interdependencies of log lines.

In order to overcome these issues, a dynamic log file anomaly detection methodology is introduced in this thesis. The procedure employs an incremental clustering algorithm that groups similar log lines within predefined time windows. Thereby, a novel clustering mechanism establishes a link between the otherwise isolated collections of clusters. Cluster evolution techniques are employed to analyze the connections between clusters from neighboring time windows and determine transitions such as splits or merges. A self-learning algorithm then detects anomalies in the temporal behavior of these evolving clusters by analyzing metrics that are derived from their developments.

A prototype that incorporates the aforementioned methodology was developed in the course of this thesis and applied in an illustrative scenario consisting of a log file containing known anomalies. The results of the evaluation were analyzed in order to identify the influences of certain parameters on the ability of detecting anomalies as well as the required runtime. The evaluation of this scenario showed that 61.8% of the dynamic changes of log line clusters were correctly identified, while the false alarm rate was only 0.7%. The ability of efficiently detecting these anomalies while self-adjusting to technological changes suggests the applicability of the introduced approach.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The modern world relies on the functioning of computer systems. Within only a few decades, digital networks have spread all over the globe, thereby disrupting existing technologies and permanently affecting economical and social structures. Nowadays, digital systems that exist in all kinds of forms and scales are omnipresent. They comprise the building blocks of the Internet and are thus deeply rooted in enabling and supporting communication between humans as well as machines. For this, dispersed webservers and other devices allow information to be accessed remotely by anyone connected to this global network. Moreover, microprocessors and sensors have recently been in the spotlight for being able to contribute to what is known as the Internet of Things (IoT), a concept that refers to the increased connectivity and interconnectivity of everyday objects. Also companies have long understood the potential benefits encompassed by these technologies. Integrating IoT in their business processes is an essential step for staying competitive within the so-called Industry 4.0.

Despite all of the highly promising benefits that can be drawn from such an interconnected world with ever expanding networks, it is important to recognize the dangers that follow along this trend. First of all, it is a difficult task to analyze and reason about the enormous amount of generated information due to limited computational power or the lack of overview required to filter out relevant pieces. Furthermore, larger and more complex networks entail the emergence of threats and novel attack vectors. Not just the amount of potential entry points becomes larger in a growing network, there is also a substantial increase of the attack surface when more complex technologies are present. This allows an attacker to infiltrate the system in more diverse ways. Additionally, the actions taken by a single individual stay easily unnoticed in the vastness of information, connections, executed operations and commands that are sent and received within the network.

Threats to computer systems appear on any scale. In the past, private or home networks as well as large company networks have often posed the target of cyber attacks. The aims thereby reach from espionage and stealing of data to more severe interceptions such as the destruction of both software and hardware. Exploits sometimes appear very immediate with consequences showing not before the system is already compromised. On the other hand, some attacks persist over a longer duration in order to infiltrate the system as deep as possible and spread the malicious piece of software to multiple systems. An example for such an advanced persistent threat is the worm Stuxnet (Mitchell and Chen, 2014) that infiltrated multiple industrial plants in 2010.

The motivation behind those attacks can be just as diverse, reaching from political or economical goals to the self-administered justice of a disgruntled employee or have no purpose at all besides the personal entertainment of the attacker. Especially the threat of an employee or another authorized person attacking a system while accessing it with their privileged accounts is difficult to mitigate. This malicious activity is usually called an insider threat (Spitzner, 2003). Cybercriminals typically operate on a more professional level and have the required experience and knowledge to circumvent existing security measures, allowing them to gain unauthorized access to networks and overtake control systems. Also attacks that are not mainly targeted at the destruction of hardware or physical properties may have negative side effects, such as the malfunction of vital processes in industrial plants. Targeted attacks that are intentionally aimed at the failure of such physical systems in order to evoke life-threatening situations, e.g., a power supply breakdown or critical failures in nuclear power plants, are regarded as acts of terrorism and have impacts far beyond the scope of a single company and its employees. Finally, there have also been several accusations of national espionage and system penetration by secret governmental agencies having technological resources at their disposal that exceeds the possibilities of individuals or groups (Cardenas et al., 2009).

Although attacks on computer systems have been existing for just as long as computers themselves, recent technological advancements and the dependencies of humans on the functioning of networks and digital devices have led to a higher severity of impact connected with each threat. As a result, the field of cyber security emerged that encompasses subfields such as cryptography, protection and legal issues. This thesis focuses on the detection of attacks on computer systems, also known as intrusion detection.

A detected attack, failure or any other problem occurring within a computer system in an industrial production site usually results in immediate action by people responsible for tracing the root of the problem in order to ensure the safety of the system and prevent further damage from occurring. Even more important, appropriate countermeasures should be introduced after proper forensic analysis in order to prevent the problem from occurring again in the future. A computer system may offer several tools to aid this procedure of finding the cause of the problem, however many of them do not dig deep enough to uncover the actual commands that were sent through the system that may give a specialist the informations that are required to thoroughly understand the incident that caused the issue.

Fortunately, the low-level console log exists for almost every system and keeps track of every single event that is carried out. Due to the fact that these console logs are designed to be human-readable, they usually contain text messages and give information about parameters and other values related to the currently running processes (Xu et al., 2009). It should be noted at this point that attempts have been made to standardize log files in order to counteract the problems that occur when logs are automatically read or parsed by a program. One of the more popular standards is the Common Log Format[1] used for web servers. Each line contains the credentials of the remote user, the request being sent, its status, the length of the transferred document and an optional date, all separated by spaces and in a specific order. Other norms describe the expected behavior of communication protocols such as the TCP/IP syslog protocol[2] that is less restrictive and allows valid log lines to take highly different forms, thereby impeding proper parsing.

Besides from debug output during software development, logs obviously aim at creating a permanent documentation of information system operations. Technicians are able to consult the historic records for reconstructing the past. For security reasons and due to the fact that pure text-based logs can be compressed and stored very memory-efficient, logs are also rarely deleted and therefore contain large amounts of information gathered over a long time. As a result, logs are frequently used for auditing purposes and their generation, preservation and protection may underlie legal regulations in specific circumstances (Kent and Souppaya, 2006). It is very likely that the source of the problem can be found when following the hints that are derived from log records related to the issue, even though this means that massive amounts of lines need to be analyzed manually. Clearly, if the log files contain all the relevant information that is required to forensically investigate a problem that happened in the past, it is also possible to monitor the log lines in real-time in order to detect any occurring problems instantaneously. Automatically triggering immediate actions after an anomaly has been detected may be able to protect the system from any adverse consequences that follow up. This is a seemingly impossible task for humans due to the high frequency in which log lines have to be processed and the cognitive abilities required for keeping track and analyzing the data. However, machine learning may fit perfectly for this task.

1.2 Problem Statement

Whenever machine learning methods are considered, the type, structure, quality and quantity of the input data immediately restricts the range of usable algorithms as most of them require the data to exhibit certain features, e.g., numeric values. Furthermore it is commonly known that machine learning is most successful if the parameters of the applied methods are optimized and fitted for the specific case at hand, but the very same settings may fail to reproduce results of a similar quality on any other input dataset. The

[1] Common Log Format by World Wide Web Consortium (W3C) available at `https://www.w3.org/Daemon/User/Config/Logging.html#common-logfile-format`, accessed 25-September-2017

[2] The BSD syslog protocol by C. Lonvick (Cisco Systems) available at `https://www.ietf.org/rfc/rfc3164.txt`, accessed 25-September-2017

largely text-based and diverse contents of log files therefore make any analysis difficult. There are uncountable different ways how log files are structured in practice and the contents of most real-world log files exhibit highly different features as they depend on the type of application, configurations defining what type of messages are logged (e.g., informative messages, errors or debug output), the verbosity of the log lines, what kind of components are placed in the system and in which way they are writing their messages to the log file.

This kind of content diversity apparent in many existing applications renders an automated analysis impossible and thus requires learning methods that provide a more flexible way of extracting relevant data out of the logs. Moreover, locating lines that contain significant words like "error" is also not enough for a thorough analysis of the system and neither is the presence or absence of certain lines sufficient to indicate problems, but rather the dynamic relationships and correlations between lines have to be considered when performing anomaly detection (Xu et al., 2009).

As previously mentioned, knowledge about the logging standard is obviously advantageous for designing an anomaly detection technique. However, this also requires the creation of a parser specifically for every type of log line and will not be able to work with lines that do not cohere with any of the predefined standards. There is therefore a need for a universal solution that is not limited to only a specific log format or logs created by a specific device such as a web server, but rather operates on any type of log file.

There exist several approaches that fulfill this requirement by employing unsupervised or semi-supervised text clustering approaches that operate independent from the structure of the log file at hand. These methods group similar log lines into a collection of clusters, i.e., a cluster map. However, the cluster maps resulting from these algorithms usually only give a static view of the data and mostly neglect any dynamic features. Wurzenberger et al. (2017) create such a static cluster map in an initial training phase and observe log line allocations to these clusters over time, however do not sufficiently take into account that a static cluster map cannot be used as a permanent template for a computer system. This is due to the fact that any system generating log lines is constantly subject to changes and therefore cluster maps generated during a specific time window often turn out to consist of highly different structures. It is therefore necessary to incorporate dynamic features into the static cluster maps.

This task is known as cluster evolution analysis. Figure 1.1 shows an example of three cluster maps generated in three different time windows. In the first time window, the cluster map consists only of a single cluster. This cluster contains a set of log lines displayed as points and is defined by a representative, i.e., a specific element marked by a star that represents the content of the cluster. In the second time window, two clusters exist, but only one of them is a descendant of the cluster from the first time window. This relationship between the clusters is marked by the arrow pointing from the original to the resulting cluster. In the third time window, three clusters exist, but two of them originate from a single cluster, thereby forming a split. It is non-trivial to determine such transitions between clusters due to the fact that log lines are non-recurring objects,

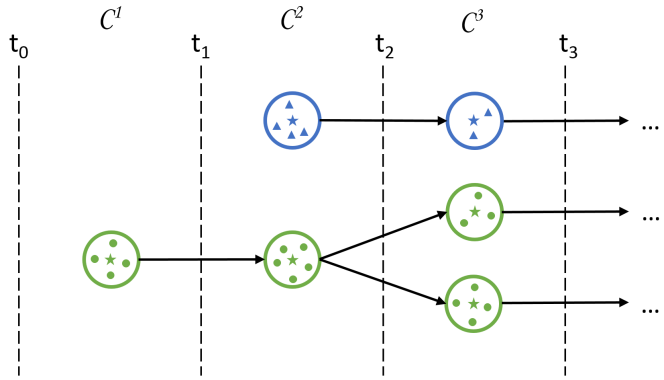

Figure 1.1: Example of cluster evolutions spanning over 3 time windows.

i.e., a log line occurs exactly at one single point in time and is never observed again due to time stamps, IDs and other artifacts in the strings. In other words, static cluster maps generated from different sets of log lines do not share any common elements and therefore render existing cluster evolution techniques useless.

Anomaly detection always relies on some kind of metric that determines whether a specific instance such as a log line, group of log lines or point in time is anomalous or not. Predefined limits are frequently used to trigger alarms for these metrics, however are not always an appropriate solution in an unsupervised setting. This is due to the fact that different systems usually show highly different behavior and also the behavior of a single system changes over time. A self-learning procedure should therefore be able to dynamically adjust to any environment it is placed into and adapt the limits for triggering alarms on its own.

Finally, an anomaly detection system that deals with all the previously mentioned issues must also exhibit a reasonable computational complexity regarding runtime and memory consumption. Due to the fact that online anomaly detection is supposed to take place in real-time, the algorithm needs to be efficient enough to process log lines faster than they appear. Furthermore, it must be ensured that the used methods are suited for the processing of streams, i.e., the runtime should scale linearly with the number of log lines and there must not be a constant growth of the required memory.

The aforementioned issues raise several questions that are relevant for performing anomaly detection based on the evolution of cluster maps. Among these are the following:

1. How can the evolution of log line clusters be mapped over time, i.e., how is it possible to find relations between two or multiple consecutive cluster maps?

2. What is an appropriate cluster representation that supports the tracking of clusters over time?

3. How is the tracking of clusters influenced by advanced transitions such as splits or merges?

4. What are appropriate measures that can be derived from the dynamic development of the clusters that give insight into the current status of the system and can be used for the detection of abnormal behavior?

5. What are suitable metrics that indicate that a new cluster map needs to be created?

6. How can appropriate values for the parameters used in clustering be estimated?

This thesis focuses on questions (1)-(4) and a thorough investigation on the remaining questions is considered out of scope. However, the findings of this work should give an idea about the relevance of the open issues by pointing out their influence on the results. Thereby it should be possible to choose reasonable parameters for practical applications and further support future work that is based on the topics of this thesis.

1.3 Aim of the Work

The research carried out in this thesis introduces the definition of a dynamic anomaly detection methodology for log files and continuous log streams. Furthermore, the theoretical concepts and the established models are implemented in a functioning prototype. The methodology encompasses the following contributions:

1. a clustering model that is able to connect log line clusters from a sequence of static cluster maps and thereby supports the detection of transitions between these clusters,

2. the definition of metrics that are derived from aforementioned transitions between clusters,

3. an anomaly detection approach that displays the security-relevant metrics as time-series and employs forecasting models in order to detect deviations from expected behavior,

4. a concept with linear runtime scalability and limited memory requirements of the whole procedure in order to ensure online processing capability and

5. an evaluation of the introduced methodology by deploying the prototype in a realistic scenario.

The main feature of the introduced approach is that contextual anomalies, i.e., log line types that do not cohere to previously gained knowledge about their average frequency of occurrence, periodicity and correlation, are detected. This extends the ability of static clustering approaches that detect highly dissimilar lines which occur only once as outliers

rather than temporal anomalies which are observed as system behavior changes over time. Moreover, the introduced approach is self-learning and does not require any previous knowledge about the structure and content of the log data. This allows the handling of complex log lines from any number of processes and components in arbitrarily formats and appearances following different standards or no standards at all.

1.4 Methodological Approach

In the first step, an in-depth investigation of existing anomaly detection methodologies is carried out. Thereby, the particular focus lies on techniques that process textual or log data from computer systems. Especially dynamic anomaly detection techniques that take temporal dependencies of the input data into account are of relevance for the topic of this thesis. This state of the art research aims at identifying key concepts that are relevant for the introduced dynamic log file analysis methodology that is based on cluster evolution techniques, i.e., methods that determine and measure the transitions between clusters. Important aspects therefore include clustering mechanisms for grouping textual data, cluster evolution methods and security metrics derived from these evolving clusters as well as prediction models for anomaly detection.

Based on the insights gained from the investigations of existing techniques, a novel anomaly detection mechanism is designed. The main part of this thesis focuses on a theoretical discussion of the used concepts and a critical reflection about their characteristics relevant for anomaly detection. These explanations are accompanied by representative examples that highlight the main aspects of the employed methods and contribute to a more intuitive understanding of the anomaly detection methodology that is introduced stepwise.

Finally, the prepared methodology is realized as a prototype that is applied in an illustrative scenario. The evaluation is carried out using a semi-synthetic data set that allows the computation of rates that measure the performance of the approach. Together with measures regarding runtime and scalability, these metrics should then indicate the overall effectiveness and determine whether the introduced anomaly detection methodology is of practical use in real-world applications.

1.5 Structure of the Work

The remainder of this thesis is organized as follows: Chapter 2 surveys existing approaches for anomaly detection. Most of the works are related to log file analysis or in other ways share a relevance with system security. Furthermore, some works that introduce the topic of cluster evolution and provide the basis for the methods and algorithms developed over the course of this thesis are outlined. Chapter 3 goes into detail about the incremental clustering algorithm that is used to efficiently create the static cluster maps for a specific set of log lines. Chapter 4 then extends on the clustering algorithm by introducing a clustering model that supports cluster evolution techniques. Moreover, an algorithm for detecting cluster transitions and the computation of security-relevant metrics are

explained in detail. These sequences of values are then used for time-series prediction in Chapter 5, where the detection of anomalies based on the forecasts as well as the correlation between time-series is investigated. The mostly theoretically discussed models are then applied within a realistic scenario in Chapter 6. This chapter also includes an in-depth evaluation that points out the effects of certain parameters and influences of the input data on the quality of the results. Finally, Chapter 7 concludes the thesis and further states suggestions and ideas for future research in this topic.

CHAPTER 2

State of the Art

The high risk posed by cyber threats has led to a massive interest in securing computer systems. Accordingly, a vast amount of research in the field of cyber security has been carried out and there exist numerous works focusing on highly diverse aspects of security. For the purpose of this thesis, the wide area of this research field is narrowed down to the subfield of intrusion detection techniques.

Intrusion Detection Systems (IDSs) are programs responsible for systematically monitoring the current state of a computer system and are used to analyze the retrieved values in order to detect indicators for potentially dangerous events taking place. They do this by observing all network traffic and raise or report alarms to the responsible human administrator in the case of a registered anomaly.

Intrusion Prevention Systems (IPSs) are more sophisticated mechanisms that usually include an IDS but are also able to take actions that are appropriate for keeping the system in a safe state during and after an imminent attack. These actions include closing connections that might have been infiltrated or stopping the execution of a detected malicious process.

Scarfone and Mell (2007) define three different methodologies that are used for the detection of incidents that are further specified for IDSs by Liao et al. (2013) and IoT devices such as cyber-physical systems (CPSs) by Mitchell and Chen (2014):

1. **Signature-based Detection**: Also known as knowledge-based detection in the field of CPSs, signature-based intrusion detection aims at identifying predefined patterns of malicious behavior. Assuming that most of the possible attacks are known and stored in an attack dictionary, this method is highly effective as it usually raises false alarms very rarely. However, as previously mentioned, the possibilities of attacks are unfortunately very diverse and novel attack scenarios may appear with changing technologies that can impossibly be foreseen at design time. If the

dictionary is known to the attacker, it is also possible that the intruders circumvent the attack vectors by adapting their methods accordingly. Maintaining the attack dictionary is therefore an essential but also time-consuming task. Furthermore, signature-based methods typically do not consider the current state of the system which could be important as most events are only posing a threat at specific times or occurring in certain combinations. Because of these issues, signature-based detection alone is generally not sufficient for a long-term threat prevention mechanism in most large-scale practical applications.

An example for such a rule-based intrusion detection system is SNORT (Roesch, 1999). SNORT works similar to a sniffer as it inspects network packet payloads and compares the content with a set of predefined rules. If there is a match with one of the rules, an anomaly is detected and an alarm is raised. Despite the previously mentioned shortcomings of signature-based detection, SNORT gained widespread popularity as it was designed to be lightweight, meaning that it operates on all kinds of systems without large installation effort and allows rules to be established very easily.

2. **Anomaly-based Detection**: Also known as behavior-based detection, anomaly detection is an unsupervised approach that learns the normal system behavior over time and is able to identify anomalies that do not cohere to the observed patterns. Contrary to signature-based detection methods, there is generally a higher amount of false alarms as outliers occur naturally from time to time without actually being caused by a malicious event. Anomaly-based detection techniques are the main focus of this work and different approaches as well as examples will be thoroughly examined in the following parts of this thesis.

3. **Stateful Protocol Analysis**: Generally speaking, this technique is based on comparing current system behavior with a predefined profile that describes universal behavior and is usually provided by the vendor of a product. Analogously, behavior-specification-based detection for CPSs requires that normal behavior of a system is defined as a model by an expert. Deviations of any kind from that expert model are reported as abnormal behavior. With signature-based detection representing a blacklist-approach that specifies all actions that are not legitimate, stateful protocol analysis is a whitelist approach and is thus able to detect attacks that do not necessarily have to be known in advance. Furthermore, this approach does not require any training phase in order to learn the normal behavior, but rather is able to operate from the point of installation. This approach is therefore highly effective. However, specifying and regularly updating a model that covers all allowed actions is a non-trivial task and requires high effort and expertise. Furthermore, the ability to keep track of the current state of the system that is required for an advanced detection of deviations is sometimes computationally extensive. Finally, there always remains a chance that malicious behavior is conduced within the limits of the profile and can thus not be detected.

2.1 Anomaly Detection

For the scope of this thesis, anomaly-based detection is the most relevant technique. Not all anomaly detection algorithms can be used for any problem at hand. Usually, the structure of the data narrows down the possible choices. For example, textual data cannot be used with numeric machine learning algorithms without prior processing, data with large sample sizes may exceed computational limitations of algorithms which scale quadratically with the amount of samples, and data without labels cannot be used with algorithms that build a model of all possible anomaly classes during a so-called training phase. Especially the latter distinction is commonly used to separate anomaly detection algorithms in the following types (Chandola et al., 2009; Goldstein and Uchida, 2016):

- **Supervised algorithms** require a set of labeled training data in order to build a model that is then used to classify new data based on the previously gained knowledge. Algorithms exist for both the discrete case, where data points are elements that can be allocated into one or more classes (e.g., K-Nearest Neighbors, Decision Trees or Support Vector Machines), and the continuous case, where the target value is numeric (e.g., Regression). As supervised algorithms always require labeled input data, evaluation of the resulting classification is generally very easy and can be carried out by using a fraction of the input data as a test set where the predicted classes are compared with the known actual classes of the samples. The output of this evaluation process is the accuracy of the model for the given test set that represents the percentage of correctly classified test samples. Furthermore, it is possible to split up the input data into an additional validation set that is used to enhance the quality of the classification by optimizing parameters. This also includes the prevention of overfitting, i.e., the situation where the accuracy of the model decreases in the general case caused by a too specific adjustment on the training data. More advanced techniques include k-fold cross-validation, a procedure where the input data is split up into k smaller parts, each of which is used once as the validation set.

  A drawback of this method is that anomalies only make up a small fraction of the input data in most real-world scenarios and this imbalance of class instances can be problematic for some supervised algorithms. Moreover, labeled data can be very difficult and expensive to gather in practice, thus often preventing the application of supervised algorithms. Especially for log files, the large number of lines that needs to be labeled contribute to the difficulties when creating or gathering suitable data.

- **Unsupervised algorithms** do not require any kind of labeled samples and are typically used for clustering large amounts of data. The approaches often aim at grouping similar objects based on their distribution, distance, density or any other measurable property. There exist several popular algorithms, e.g., K-Means or Self-Organizing Maps. As no labeled training data is required, unsupervised algorithms are widely applicable. In addition, unsupervised algorithms implicitly

assume that outliers only make up a small part of the input data, which makes them very well fitted for anomaly detection. Drawbacks of unsupervised algorithms include difficulties regarding both parameter optimization and evaluation which may require iterative manual work due to the lack of labels that could be used for comparisons and estimations of cluster quality.

- **Semi-Supervised algorithms** contain labels only for some samples of the training data, for example, only one discrete class is labeled and all the other classes are unknown. This case is likely to occur in a real-world scenario and thus semi-supervised algorithms are essential in practice. While for system analysis there could be a possibility to gather training data in a secured environment that is assumed to be anomaly free and can thus be classified as "normal", there is typically hardly any data that contains known anomalous behavior and classes corresponding to the type of anomaly. Nevertheless, this starting position is often sufficient for anomaly detection in system security, due to the fact that it is only important to identify that an anomaly occurred rather than determining its exact type. However, due to the fact that systems change over time, data that is known to be anomaly-free would have to be gathered regularly in order to update the data base provided for the machine learning algorithms. This is a time-consuming task and thus difficult to apply in practice.

Anomaly detection bases its functioning on the fact that any malicious event manifests itself in an observable or measurable way. As it is unusual that attacks occur, this manifestation is expected to stand out from the normal system behavior. Such suspicious events that do not cohere with the overall behavior indicate anomalies. The following types of anomalies are differentiated (Chandola et al., 2009):

1. **Point Anomaly**: This is the simplest form of anomaly. A point anomaly is a single object that does not follow the overall structure of the data, i.e., it is highly different to all the other data points. When representing the data in an appropriately dimensioned space, this point is located far off the other points in one or more dimensions and thus point anomalies are also called outliers. In cluster analysis, a point anomaly would not be allocated to any existing group due to its high distance to all the other samples and would thus form its own cluster where it remains alone.

2. **Contextual Anomaly**: An instance that is only considered anomalous if it appears in a specific context is called contextual or conditional anomaly. The context could be defined by parameters or the state of the system which indicate what kinds of log messages can be considered normal or anomalous. Also the current time can be used to detect contextual anomalies, e.g., a high frequency of recorded network connections within a company can be considered normal during daytime but the same amount may be suspicious if it occurs in the middle of the night. By

implication, time-series analysis is a popular methodology used for the detection of contextual anomalies.

3. **Collective Anomaly**: An anomalous group of related instances is called a collective anomaly. It should be noted that the instances forming the group are not necessarily abnormal themselves, but only the group as a whole is considered anomalous. When considering system security, attacks are typically performed in a sequence of steps that all manifest themselves in the system's log. While each single of these lines may not be especially suspicious, the combination and particular order of them can be a clear indicator for an attack. Correlating all events with each other in an efficient and effective way is usually a non-trivial but necessary task in order to detect collective anomalies.

A fundamental difference of anomaly detection algorithms is whether the time dimension is included in the learning process, i.e., whether the temporal development of relationships between events are considered to be a potential indicator of dangerous behavior or whether the system is analyzed in every time step independent of what happened in the past or will happen in the future. In the following, an overview about current approaches for both possibilities is given.

2.1.1 Static Anomaly Detection Techniques

Given that determining whether an occurring event is malicious or not is typically a decision under uncertainty, a natural approach is to apply appropriate statistical methods to this problem. Kruegel and Vigna (2003) analyzed log files containing HTTP queries by computing probability values for several attributes that are regarded as indicators for anomalous behavior, for example, attribute presence, length, order and structure. Another highly popular way of including probabilities to a decision problem is by utilizing the Bayes Theorem, a powerful method that takes the conditional probabilities of elements such as textual words being present or absent from certain classes into consideration. This approach proved itself highly successful when applied for spam filters in the past (Metsis et al., 2006). Similarly, Bayesian statistics are used for the detection of anomalies or attacks in network traffic protocols of computer systems. The Bayesian approach has been carried out by Amor et al. (2004) and its performance was compared to that of a decision tree classifier. It was found that while both approaches show competitive results and their respective abilities of detecting anomalous behavior largely depends on the type of attack at hand, the computation time of Bayesian methods is generally lower. Furthermore, the effectiveness of a Bayesian classifier was enhanced using a preceding K-means clustering by Yassin et al. (2013). Data integrity attacks on a CPS were also detected by utilizing the more complex method of learning Bayesian networks. As shown by Krishnamurthy et al. (2014), this approach does not only successfully identify anomalous events in network data, but is also able to track the anomaly to its source, thus determining whether it was an attack or a physical breakdown that raised the alarm.

The hosts and connections of computer systems can be seen as the nodes and edges of a graph. Akoglu et al. (2014) point out several advantages of anomaly detection using graphs, including their natural way of approximating real-world networks and their ability to capture long-range correlations between nodes. Graph-based anomaly detection aims at identifying abnormal nodes, edges or substructures and can be based on several measures, including global metrics like the distance and distribution of nodes, the depth of the graph, node-centric measures such as the in- and out degree or neighborhood-based metrics. In another example, Noble and Cook (2003) make use of an algorithm called Subdue that is able to iteratively discover and replace patterns in a graph, thus revealing infrequent and potentially anomalous substructures. A problem that arises when trying to apply graph-based anomaly detection on textual or log data is the high complexity, as edges between any of the text fragments can be computed, thus exponentially increasing the required computation time. Graph-based methods should therefore be favored in cases where each node only has connections to a small fraction of the total amount of nodes, e.g., in the case where connections between hosts are observed.

Many popular text classification methods are focusing on creating patterns or signatures of anomalous events and log lines that are then used to group similar or related attacks together. As stated by Vaarandi (2003), log files usually have no standardized format and thus many association rule algorithms fail to work. As a solution, the authors introduce the clustering algorithm SLCT that is able to generate patterns by observing frequent words and their respective positions in each line by a single run through the log file. By allocating the lines to the generated patterns in a second run through the log file, SLCT identifies outliers that do not match any of the previously created rules and detects them as anomalies. Even though SLCT is known to terminate very fast even for large input data, the quality of the results largely depend on the often highly sensitive parameter settings (Stearley, 2004) that easily cause the detection of an overwhelming number of false positives. Furthermore, all the cluster templates must be stored for the second run, which possibly raises memory problems for large log files. Another rule-based approach was introduced by Breier and Branišová (2015), where log lines are transformed into a binary format and compared with predefined patterns of normal behavior. Outliers that do not match any of the rules are then reported as anomalies. This approach uses parallel processing and employs MapReduce in order to keep the runtime at a minimum.

A well-known way of measuring the distance between strings from an unstructured text file is comparing their respective n-grams. Juvonen et al. (2015) therefore projected log lines according to their common n-grams in a high dimensional space and identified outliers as lines that deviate too much from the computed average. In order to tackle the curse of dimensionality, i.e., the problem of sparsity occurring when data points are placed in high dimensional spaces, three different dimension reduction techniques were used for experiments and it was found that both Random Projection and Diffusion Maps should be favored to Principal Component Analysis. It must be noted that the temporal correlations inherent to log file lines can negatively influence the results of anomaly detection techniques based on dimension reduction (Brauckhoff et al., 2009).

2.1.2 Dynamic Anomaly Detection Techniques

Although the order of the input data may have an influence on the results of a static anomaly detection technique, many of the algorithms do not fully make use of the temporal dimension, e.g., by simply ignoring the timestamp attached to each log line. However, by integrating this information in the anomaly detection process it is possible to derive new insights such as long-term trends and correlations between clusters that would otherwise remain hidden. For example, a special type of log line that is observed once precisely at the start of every hour over a long period of time is most probably generated by a scheduled task and it could thus be predicted at what time the line should appear again in the future. If the line however is delayed, suddenly skips one of those scheduled points in time or changes its periodicity to a different interval than one hour, an alarm should be triggered as this may indicate anomalous system behavior. More sophisticated timing restrictions include log lines that consistently appear after some time when a specific event occurs. However, such delayed relationships can easily be overseen when dealing with a large number of log lines that occur with varying frequencies. Especially when the root of some problem needs to be detected, knowledge about these temporal correlations and causal relationships is essential to rapidly detect the origin of the problem. The causal relationships can also be deeply nested and there is the possibility that multiple sources contribute to the outcome of a specific log line which increases the difficulty of detecting the cause of a problem (Rouillard, 2004).

Obviously it is always possible to carry out any static Anomaly Detection technique in every possible time step in order to obtain results that are based on the elapsed time. However, the nature of log data counteracts such a procedure for several reasons. Firstly, time stamps of most logs contain seconds or even more precise measurements, thereby causing that an enormous amount of steps has to be taken into account. Algorithms such as cluster analysis are typically complex methods that require high computational effort and therefore cannot be carried out for every single time step. Secondly, the fact that events generally do not appear in regular intervals has to be considered, meaning that there could be some period of time where log lines are generated at a much higher rate than at other times. Finally, it can be assumed that adding one single line or only a few lines to the current clustering does not have a large influence on the formation of the clusters, and it is therefore not necessary to completely repeat the clustering procedure from the start, but rather incrementally add the newly arriving data points to the existing clusters and observe the features of the clusters over time.

It is important to differentiate between two cases where the term incremental clustering is used in literature. On the one hand, this term is used to describe a sequential data flow that is continuously clustered. Log lines form such a stream of incoming strings and hence this thesis will only focus on that kind of clustering. On the other hand, incremental clustering is also used to describe the scenario where the same objects are observed over multiple time steps (Xu et al., 2014). It was already mentioned that log lines are non-recurrent objects and are thus not suited for such methods.

When features are measured over time, ordered sequences of values are generated. A popular way of describing these time-series is the combination of autoregressive (AR) and moving average (MA) models into so-called ARMA models. Further extensions exist, such as ARIMA models for non-stationary time series or SARIMA models that also include a seasonal component. A related but less complex modeling technique that also adds trends and seasonal changes to the predictions is the Holt-Winters model. Outlier detection is realized by forecasting the values of a time series and computing the deviations from the actually measured data, where deviations that exceed a certain threshold indicate anomalous behavior.

The already mentioned graph-based anomaly detection is not limited to static networks. Pincombe (2005) uses an ARMA process for modeling the temporal behavior of the graph consisting of TCP/IP connections between users. The author optimized the ARMA model for several different graph distance metrics and discovered that only some of those metrics appropriately detected the known anomalies in the data. Bilgin and Yener (2006) add that one metric is usually not sufficient to describe the graph as a whole and anomaly detection algorithms should therefore be based on a combination of several features.

Neural networks are a popular choice for learning patterns in data. Cortez et al. (2012) compare an ensemble of multilayer perceptrons with ARIMA and Holt-Winters time series models. They found that neural networks show comparable results and are able to surpass the time series models regarding the runtime. Another comparison study by Hill and Minsker (2010) also uses a multilayer perceptron, an AR model as well as a modified k-Nearest Neighbors classifier for investigating the influence of detected outliers. It was found that leaving the outliers in the data may have a negative influence on the following forecasts, especially for predictors that place a high weight on the recent data points. There is therefore a need to mitigate the influence of these points and it is suggested to replace the anomalies with the values predicted by the model.

A more complex neural network is the Long Short Term Memory Recurrent Neural Networks (LSTM-RNN) which is able to learn temporal dependencies of the input data. Goh et al. (2017) simulate a CPS attack by injecting wrong sensor information into the system for different time durations. The neural network was trained with normal system behavior and was able to detect a high number of simulated attacks even when they were targeted to different processes of the CPS. Fiore et al. (2013) use a special type of neural network called Restricted Boltzmann Machine to experiment on how differences between the datasets used for training and testing affect the quality of the results. It was shown that using a different network for training is likely to decrease the effectiveness of the anomaly detection on the test set.

In a similar manner to a rule-based approach, Fu et al. (2009) extract the log keys of each log line by omitting parameters so that only the non-variable parts of the log line remain. Contrary to SLCT, clustering of the lines is then carried out by comparing string metrics rather than pattern matching. A finite state automaton is created and deviations of both the transit time from one state to another as well as the circulation number for all the loops are considered as measures to detect anomalous behavior. A similar approach for

retrieving information from log files was carried out by Xu et al. (2009), where templates for most of the lines were created by applying static source code analysis to the program generating the log lines. The state transitions within a time window were observed using the subspace method and the results of the outlier detection were enhanced by TF-IDF weighting of the message counts.

He et al. (2016) introduce an algorithm that also creates log line templates and clusters the lines within time windows. For each window, an event count matrix is filled that keeps track of the amount of occurred log line types. The authors experiment with different machine learning techniques that learn expected behavior from such an event count matrix and are able to detect anomalies by comparing the learned model with event count matrices generated from the currently processed lines. Additional to experimenting with time windows of a fixed length, a sliding window approach and session identifiers were used to split up the log file. A system that changes its behavior over time requires that the learned model is updated by regenerating the templates regularly.

A more generic approach that does not create templates is introduced by Andreasson and Geijer (2015). Instead of matching the patterns of some generated rules, this approach considers string metrics and n-gram matching for grouping similar types of lines together. The log files are analyzed and summarized by message counts and word occurrences for every hour, thus creating a convenient and not overwhelming summary about the system behavior over time. A normal distribution of those retrieved statistics is assumed and high deviations far outside of the Gaussian curve are reported as anomalies during the detection phase.

Finally, there also exist several commercial solutions that offer log file analysis. Besides allowing visualization and providing the ability to search through log files, Logentries[1] also applies real-time anomaly detection on logs. The program generates a profile as a baseline of normal behavior and raises alarms if deviations from that profile occur. Furthermore, anomalies are raised when log lines corresponding to scheduled events do not take place as expected. However, similar to other previously mentioned approaches, Logentries requires the definition of parsers for all line types in order to extract numeric values that are then compared with predefined thresholds.

2.2 Cluster Evolution

While a large amount of research has been carried out in the field of time series analysis and its subfields clustering (Silva et al., 2013; Esling and Agon, 2012; Khalilian and Mustapha, 2010) and anomaly detection (Sperotto et al., 2008; Thottan and Ji, 2003; Chin et al., 2005; Gupta et al., 2014) in time series, many of those approaches are not directly applicable on cluster evolution analysis as they focus on input data that consists of one or more features that are measured over time, i.e., for every feature there exists a value in every discrete time step. Though being a promising start as, for example, the

[1] Logentries available at `https://logentries.com/`, accessed 25-September-2017

sizes of log message clusters can be seen as features that are observed over time, a more sophisticated analysis of the clusters is required in order to keep track over their temporal changes. Hence, the field of cluster evolution specifically aims at identifying trends in cluster developments, which can in turn be used to detect abnormal cluster behavior.

The purpose and possible insights that can be derived from a clustering were already explained previously. In general, clustering algorithms are designed for a static view of the underlying data, i.e., a set of data points is considered to generate the cluster map for a specific time span. Although the ordering of those data points can have an influence on the number and structures of the resulting clustering, most cluster techniques are not specifically designed to support dynamic changes that are caused when inserting new data points to an existing cluster map, but rather require a complete reformation of all clusters by starting the algorithm all over again. As a solution, incremental cluster methods are able to dynamically add any number of incoming data points by either allocating them to one of the existing clusters or declaring them as outliers if the distance to the nearest cluster exceeds a certain threshold.

The incremental approach has been applied for log file analysis on systems with a highly predictable behavior and a large number of repeating sequences. It was found that the methods are able to successfully cluster the log lines that are generated by normal system behavior while detecting outliers that potentially represent anomalies (Wurzenberger et al., 2017). However, for modern computer systems and networks the assumption about a steady system behavior is not necessarily valid due to several reasons. Any component within a network that contributes to the generation of log lines is subject to modifications or replacement. These actions can change the format or content of the logged lines that will then not be clustered into the same group anymore, causing alarms to be raised as all of the lines will be detected as outliers although representing normal system behavior. This also causes that no more lines will be allocated to the original cluster which should also be detected as an abnormal behavior that should trigger some kind of alarm. In a more complex case it could also be possible that only a fraction of the lines that would be allocated to a cluster fall outside of that group and become outliers. These and even more complex scenarios that will be investigated in detail in the following chapters of this thesis could easily be imagined within a large-scale network that involves large numbers of users and programs. Furthermore, static cluster analyses generally have no way of reacting to periodical changes that almost always occur in real-world networks. As a simple example, it can be assumed that network traffic and server accesses are much higher during the day than during the night as most people are working on daytimes in their offices, while only some automatically scheduled programs operate non-stop. Another example is the decrease of human network accesses on the weekends compared to weekdays.

The issues that arise when performing cluster evolution analysis are manifold and most works focus only on a certain aspect of cluster evolution. Given two cluster maps from two different time windows, it is usually the essential first step to identify which cluster from the former map transformed into a corresponding cluster from the later map. This

task is referred to as *cluster tracking* and obviously has to cope with changes in cluster composition, e.g., the allocated members are not necessarily identical in both time windows. In the next step, *cluster transitions* that are based on the changed cluster structure have to be identified. They are usually divided into external transitions that include emergences, disappearances, splits and merges and internal transitions that include changes in size, spread and location (Spiliopoulou et al., 2006). Finally, appropriate *evolution metrics* have to be designed that allow a representative quantification of the ongoing cluster dynamics and form time-series that are applicable for anomaly detection. The detected anomalies may include any unexpected behavior of the time-series, such as rapid changes in size, sudden disappearances or deviations from long-term trends.

Especially short-term anomalies likely influence the quality of the cluster map in the following time step as objects could be compared with outliers that are not representing the systems normal state correctly. In order to overcome this issue, smoothing can be applied to the evolution of the clusters in order to maintain a steady and non-noisy cluster development. Chi et al. (2009) combine the snapshot cost which captures the quality of the current snapshot with the temporal cost which captures the alignment with previous snapshots from historic data. They introduce two frameworks which optimize this value by either focusing on preserving cluster quality or preserving cluster memberships and use both K-Means and spectral clustering as underlying algorithms. Xu et al. (2014) build upon this approach and apply an evolutionary clustering algorithm in order to approximate the distance matrix of all data points.

Any kind of clustering technique may be used for the application of cluster evolution, however, some methods may be better suited than others. Chakrabarti et al. (2006) point out that clustering algorithms should take historical data, i.e., the results of clusterings from previous time steps, into account in order to ensure that the differences between cluster maps are kept at a minimum. Otherwise, even very similar data can lead to highly diverse cluster maps, thereby impairing the quality of the cluster evolution analysis results. Moreover, the time window is required to be chosen appropriately in order to ensure that the generated clusters actually represent the system consistently. As a solution, a hierarchical approach as well as an algorithm based on K-Means that both put emphasis on alignment with historical cluster maps and snapshot quality are introduced in that work.

Tracking clusters of a set of points moving on a usually low-dimensional coordinate system finds several important applications such as GPS and is closely related to cluster evolution techniques. Jensen et al. (2007) introduce a dissimilarity metric based on the movements of objects and a cluster feature allowing incremental updates and compact summarizations of cluster properties regarding size, location and movement. By employing their cluster feature it is possible to track any number of moving clusters and also determine splits and merges that may emerge. When a Gaussian model is assumed for each cluster, Bayesian methods are able to capture the movements of an evolving Gaussian mixture model over time (Carmi et al., 2009). As in the case of log lines it is generally neither possible to assume a Gaussian distribution nor that clusters continuously move around in patterns

or constant velocity, but rather that clusters normally remain stable and any deviation from their typical state is an exception. Therefore these approaches are not perfectly suited for the purpose of cluster evolution in the context of log file analysis.

It should be noted that an alternative approach could make use of sliding window functions that may allow a more sensitive analysis of the cluster properties. However, such approaches typically require more complex mathematical formulations and procedures. A sliding window technique for cluster evolution analysis is introduced by Zhou et al. (2008) where a data structure called Exponential Histogram of Cluster Features stores and updates the cluster properties over time. There also exist incremental methods that are able to dynamically update location and spread of clusters when new samples are added and further support merging and splitting of clusters (Lughofer and Sayed-Mouchaweh, 2015).

While the goal of outlier detection in standard cluster analysis is to find single lines that do not fit into one of the formed clusters, outlier detection based on cluster evolution seeks groupings in data that exhibit trends in their properties such as their size which could indicate anomalies. Taking this one step further, it could be interesting to observe the behavior of the objects that form a community within that cluster. The term community in this context is generally used when cluster analysis is performed on data based on social networks and related fields. Gupta et al. (2014) differentiate between Evolutionary Community Outliers where objects swap their surrounding community and Community Trend Outliers where objects do not follow the trend of their community and suggest algorithms for detecting both of these types.

An overwhelming amount of literature focuses on graph-based cluster evolution and analysis (Chan et al., 2008; Bilgin and Yener, 2006; Asur et al., 2009; Lee et al., 2014; Bródka et al., 2013; Falkowski et al., 2006). Graph theory has a strong theoretical basis and although some ideas could be transferred to a more general spatio-temporal data analysis, most of these works cannot be directly applied to cluster evolution using log files as an input stream. While for graphs, the appearance and disappearance of connections between nodes pose the essence of the analytical models, log file cluster evolution mainly focuses on the distance between clusters and objects. Furthermore, all log lines can be related to each other via some kind of similarity metric, meaning that graph-based models would always have to deal with a complete graph. This can obviously cause an issue in computational complexity due to the exponential growth of edges for an increased amount of nodes. As a solution, all the edges that represent a connection between two log lines can be removed if the distance exceeds a certain threshold, i.e., if the lines are not similar enough. In addition, the weights of the remaining edges could be omitted, leaving a binary graph behind that would allow the application of graph-based models in an efficient way. The influence of these simplifications on the quality of a clustering and community evolution analysis has not been investigated so far. A large fraction of the applications in this area focuses on dynamic social networks that consist of people and their relationships to each other as those networks can adequately be represented as a graph.

While all the mentioned algorithms only take properties based on the location of the data points into account, clusters that were created using log lines do not only consist of their members but also have additional properties that can be used to track them from one time step to the other. For example, when applying incremental clustering, one line is used as the representative of that cluster. While it is not guaranteed that a line with the very same content is selected as the representative again when the subsequent cluster map is generated, it is likely that the line is somewhat related and a similarity score can support the identification of the correct cluster. Other clustering algorithms like SLCT may generate a pattern for each cluster that can be employed for cluster generation and evolution analysis analogously, although it should be noted that pattern matching typically does not return a numeric similarity score but rather a boolean that indicates whether the line matches the pattern or not.

Creating visualizations of cluster evolution is a non-trivial task and only few appropriate solutions exist. A rather simple possibility is to display an ordered set of static images, each representing a specific state in time. A more sophisticated technique that was developed for spatio-temporal data is the space-time cube (Andrienko et al., 2003). In this method, a 2-dimensional field is complemented with an additional time dimension that shows the trajectories of the observed objects. For smaller networks that contain only few clusters, Vehlow et al. (2015) suggest so-called Alluvial Diagrams that are able to display transitions such as splits and merges by connecting ordered graphical summaries of the clusters at each time step with flow-like links. The sizes of the flows allow fast comprehension of the amount of objects that were split or absorbed from each cluster and color-coding aids the traceability of each cluster. A different approach to dynamic cluster visualization is animation. The tool GapMinder has become a popular way of displaying cluster data as it is able to dynamically depict several features at once by employing two numerical axes as well as the sizes and colors of the clusters and their respective changes over time (Rosling and Zhang, 2011). The advantage of this technique is that data is interpolated between the time steps, allowing a viewer to track single clusters in a continuous movement as well as to observe overall emergent phenomena, e.g. common trends of clusters or interdependencies within groups of clusters.

CHAPTER 3

Clustering

There exist enormous amounts of machine learning algorithms that are deployed in highly diverse application areas. This makes it difficult to maintain an overview of clustering algorithms that are suitable for the problem at hand. With most techniques only having minimal requirements on the data that is fed into the algorithms, it may be tempting to use the unprocessed data on the first available algorithm in order to solve a specific task. Not surprisingly, while the output of the algorithm may be somewhat predictable, the results are usually far from perfect and for that reason data scientists spend a lot of time optimizing the results by parameter tuning and comparing alternative approaches. Applying machine learning in order to solve a problem is not only a tedious task that requires detailed expert knowledge about the problem domain and machine learning itself, but also takes several rounds of iterative development and enhancement until an ideal solution has been found. Knowledge about the benefits and drawbacks of different types of algorithms is thus essential for initially choosing a correct learning model.

3.1 Requirements

In the case of log line clustering, several crucial characteristics need to be considered before a clustering algorithm can be employed or developed. The most important of these features are the following:

1. **String metrics**: Many machine learning techniques are centered on numbers and especially clustering almost always requires numeric values, probabilities or real-space coordinates. The reason for this is that it is much easier for a computer to find patterns or similarities in numbers than in other types of attributes. While most measurable data can be expressed numerically and many other pieces of information such as images or audio files can be converted to a collection of numbers, the strings appearing in log files usually cannot easily be transformed into a real-space. Clearly,

the numeric ASCII values could be chosen to represent each character in the string as a vector. However, computing the similarity between these vectors may not give an appropriate result as there is no reason to assume that characters closely located in the ASCII table, e.g., “A” and “B”, are more similar than characters far apart, e.g., “A” and “z”. A solution to this problem can be found in string metrics that are able to numerically assess the similarity or distance between two strings. An in-depth description of the mechanics behind string metrics will be given in the following sections.

2. **Incremental capability**: Many machine learning algorithms are only used with fixed-size datasets. This means that there is a finite amount of samples in these datasets and that the algorithm terminates once all elements have either been used for training or were classified during the evaluation phase. Practical applications of log line clustering algorithms however typically require an incremental procedure that is able to handle a stream of data containing potentially infinite incoming lines.

3. **Linear computational complexity**: Related to the problem of an endless stream of incoming data is the requirement for a linear complexity. There exist cluster algorithms that are based on comparing each element from the dataset with each other element in order to find similarities that are then used to form groups. Procedures with a complexity that exceeds linear scalability with respect to the number of lines however can impossibly be employed as the time required for the computations would ever increase, up to a point where the lines cannot be processed in real-time anymore. Due to physically limited resources it is usually not possible to upgrade the available computational power to arbitrarily high levels and thus such approaches are practically not feasible for log file anomaly detection.

4. **Limited memory storage**: Some algorithms may require to store all samples in-memory during clustering. Due to the immense amount of log lines and the previously mentioned continuous stream of incoming data, only a part of the total number of lines can be held in memory at a time. For most applications of data stream clustering, the most recent data is considered more important as it represents the most current known system behavior. Therefore, historic values that exceed a certain age should be removed from the in-memory storage and possibly saved on a hard disk in order to keep records of the conducted operations.

5. **High-performance**: The rate at which log lines are written to the log file is strongly depending on the system at hand and the number of components and programs contributing to the log file. In some practical applications, the average rate at which log lines are produced can be enormous and the algorithm must be able to keep up with that in order to ensure real-time detection capability. As long as the complexity only scales linearly with respect to the number of processed log lines, increases of average occurring frequencies can usually be compensated by increasing the computational power.

In order to tackle these issues, the used clustering algorithm relies on several components that allow the comparison of strings and pay attention to scalability and the overall performance. These tools comprising the algorithm will be explained in the following.

3.2 Word-based Matching

One way to determine the similarity of any two given strings is by computing the number of words that they have in common. Rather than splitting up strings at special characters such as spaces, it is possible to split them in regular intervals forming identical-sized word fragments. Each fragment consists of a sequence of consecutive characters of a predefined length n and is thus known as n-gram or shingle. For example, the word "string" consists of the 2-grams "st", "tr", "ri", "in" and "ng" and of the 3-grams "str", "tri", "rin" and "ing". With $|s|$ denoting the length of string s, the number of n-grams that can be formed from s is $|s| - n + 1$. It should be noted that characters appearing in the middle of the string appear in more n-grams than characters at the beginning or the end, e.g., the character "r" appeared in the 3-grams "str", "tri" and "rin" in the previous example, while the letter "s" only appeared in "str". If this behavior is not desirable, $n-1$ identical characters can be inserted before and after the string. The mentioned example would therefore be transformed into "##string##" and would result in the 3-grams "##s", "#st", "str", "tri", "rin", "ing", "ng#" and "g##", where every character appears with the same frequency. For convenience, this feature is omitted from following examples.

The Dice coefficient is an index that measures the similarity of two strings a and b by counting the amount of n-grams shared by the strings and divides this number by the total amount of n-grams in both strings. With $T_n(s)$ representing the set of all n-grams of string s, the Dice coefficient $dice \in [0, 1]$ is expressed as

$$dice = \frac{2 \cdot |T_n(a) \cap T_n(b)|}{|T_n(a)| + |T_n(b)|} \tag{3.1}$$

It is pointed out by Kondrak (2005) that not all similar strings necessarily share a high number of n-grams which can lead to a disproportionate low score. Furthermore, the Dice coefficient does not take the position of the n-grams into account, therefore also strings with many matching n-grams that are however in completely different order will lead to a rather high similarity score.

Short Word Filters (SWFs) are a more sophisticated approach that employ n-gram matching and define a threshold Θ of required n-grams that must be identical in both strings a and b in order to consider them as similar. For a given threshold $\delta \in [0, 1]$ that defines the percentage of desired similarity, the threshold of required matching n-grams is defined as

$$\Theta = min(|a|, |b|) - n + 1 - (1 - \delta) \cdot n \cdot min(|a|, |b|) \tag{3.2}$$

Note that as already mentioned before, $min(|a|, |b|) - n + 1$ is the amount of possible n-grams in the shorter string. A detailed proof of the expression is stated by Ghodsi et al.

(2011). Two strings a and b are thus considered similar if the number of shared n-grams including duplicates is not smaller than the computed threshold, i.e., $|T_n(a) \cap T_n(b)| \geq \Theta$.

Considering 2-gram matching with $\delta = 0.8$ for the example strings "string" and "strain", the required amount of matching n-grams is $\Theta = 6 - 2 + 1 - (1 - 0.8) \cdot 2 \cdot 6 = 2.6$, i.e., there have to be 3 matching 2-grams to fulfill the requirement. With the shared 2-grams "st", "tr" and "in", this requirement is met and thus the strings are considered a match. The procedure can be carried out analogously for any n and any δ with different results.

SWFs have successfully been used for clustering large amounts of biological sequences such as genomes or proteins (Huang et al., 2010; Li et al., 2001, 2002). Their advantage is that determining whether short sequences of characters are identical is much faster than comparing two very long strings. This effectively reduces overall computation time by filtering out strings that most probably do not match before applying a computationally intensive method for precisely determining the similarity between the strings.

3.3 Alignment-based Matching

Other than word-based approaches, alignment-based methods allow a more precise computation of the similarity between two strings. Computing the similarity based on the alignment avoids the previously mentioned problems of n-gram matching, i.e., issues related to the lack of order when intersecting sets of n-grams. This is accomplished by working on single characters rather than sequences of characters. The aim of alignment-based matching is to count the number of operations that are required to transform one string into the other. Existing metrics typically consider up to four different operations. These are:

1. **Insertion**: Adds one character at any position in one of the strings.
2. **Deletion**: Removes one character at any position in one of the strings.
3. **Substitution**: Replaces one character by any other character in one of the strings.
4. **Transposition**: Swaps two adjacent characters in one of the strings.

There also exist a number of string metrics that consider all or some of these operations, but weigh them differently. One of the most basic metrics is the Hamming distance that only takes substitutions into account. As no characters are added or removed, it is a requirement that both strings have the same length in order to be reasonably compared. Each substitution has a cost of 1 and the sum of all costs required to transform one string into the other is the Hamming distance. For example, transforming string "string" into "strain" requires substituting the last three characters, thus the Hamming distance is 3.

A more sophisticated and wide-spread string metric is the Levenshtein distance that is also frequently known as Edit distance and supports insertion, deletion and substitution.

The metric weighs all of them equally so that each applied operation increases the total cost by 1. Considering the strings “string” and “strain” as an example, the following sequence of operations transform one string into the other:

1. “string” → “straing” (insert “a” between positions 3 and 4)
2. “straing” → “strain” (delete “g” from the last position)

As only two operations are required, the Levenshtein distance is 2. Note that there usually exist multiple ways to successfully carry out the transformation, but only the sequences of operations that minimize the total distance are relevant. In the case of the strings “string” and “strain”, there is no other possibility to transform one string into the other using less than 2 operations. In general, the Levensthein distance $Lev(a, b) = lev(|a|, |b|)$ between the two strings a and b is computed iteratively according to the following rules (Hyyrö, 2003):

$$\begin{aligned} lev(i, 0) &= i \\ lev(0, j) &= j \\ lev(i, j) &= min \begin{cases} lev(i-1, j) + 1 \\ lev(i, j-1) + 1 \\ lev(i-1, j-1) + I(a_i \neq b_j) \end{cases} \end{aligned} \tag{3.3}$$

where $I(a_i \neq b_j)$ is the indicator function that is 0 if a_i, the i-th character of a and b_j, the j-th character of b match, and 1 otherwise. Note that the choices in the minimum correspond to the operations, i.e., $lev(i-1, j) + 1$ corresponds to a deletion from a, $lev(i, j-1)+1$ corresponds to an insertion in a and $lev(i-1, j-1)+I(a_i \neq b_j)$ corresponds to a substitution if the characters mismatch and no operation otherwise. This formula also shows that each occurring operation increments the total cost by 1.

For a better understanding, this recursive procedure is often displayed as a $(|a|+1)\times(|b|+1)$ matrix D that is filled out row-wise. Following the formula, the first row and first column are set to $D_{i,0} = i$ and $D_{0,j} = j$ independent from the content of the strings. The remaining fields are then computed by $D_{i,j} = D_{i-1,j-1}$ if $a_i = b_j$ and $D_{i,j} = 1 + min(D_{i,j-1}, D_{i-1,j}, D_{i-1,j-1})$ otherwise. The resulting Levenshtein distance is found in the bottom-right corner of the matrix. The filled-out matrix for the two example strings is shown in Figure 3.1 [1]. The path of least cost is marked with yellow and leads to the resulting distance of 2.

It is further noteworthy that the recursive formulation of $lev(i, j)$ also represents the distance between the substrings $a_1a_2...a_i$ and $b_1b_2...b_j$. This division of the problem into multiple subproblems is used in dynamic programming. In its basic formulation, the complexity in both memory and time for computing the Levenshtein distance lies within

[1] Table created using the Levenshtein Demo accessible at `http://www.let.rug.nl/kleiweg/lev/`, accessed 20-December-2017

		s	t	r	a	i	n
	0	1	2	3	4	5	6
s	1	0	1	2	3	4	5
t	2	1	0	1	2	3	4
r	3	2	1	0	1	2	3
i	4	3	2	1	1	1	2
n	5	4	3	2	2	2	1
g	6	5	4	3	3	3	2

Figure 3.1: Example for the computation of the Levenshtein distance between two sample strings.

$\mathcal{O}(|a| \cdot |b|)$. However, there exist numerous optimizations that aim at reducing the required amount of memory storage or the computational runtime. For example, Hirschberg's algorithm takes advantage of the fact that the i-th row of D is only depending from the $(i-1)$-th row that was just computed before and is thus able to reduce the required memory by a large amount (Masek and Paterson, 1980).

In general, a function $d(a, b)$ that returns the distance between two strings a and b is called a metric if it possesses the following properties (Li et al., 2004):

1. **Non-negativity**: The distance between any two strings a and b must not be negative, i.e., $d(a, b) \geq 0$.
2. **Identity**: Two strings a and b yield a distance of 0 if they are identical, i.e., $d(a, b) = 0$ if $a = b$.
3. **Symmetry**: The distance from string a to b is identical to the distance from b to a, i.e., $d(a, b) = d(b, a)$.
4. **Triangle Inequality**: The distance between two strings a and b is shorter or equal to the sum of the distances from a and b to any other string c, i.e., $d(a, b) \leq d(a, c) + d(b, c)$.

The Levenshtein distance follows these properties and is thus considered a metric. While two identical strings yield a Levenshtein distance of 0, in the case of two completely mismatching strings the Levenshtein distance will result in the length of the longer string. This property is used to normalize the distance function to the interval $[0, 1]$ which is a more comparable metric as it shows the similarity of any two strings independent from their lengths (Yu et al., 2016). The expression for computing the normalized Levenshtein distance is

$$d_{Lev} = \frac{Lev(a, b)}{max(|a|, |b|)} \tag{3.4}$$

A perfect match of two strings using the normalized function results in a distance of 0 and a total mismatch yields 1. Some applications may require a similarity metric rather than a distance metric, i.e., 0 representing a mismatch and 1 representing a match of two strings. A normalized distance function can always be converted into a normalized similarity function by

$$s_{Lev} = 1 - d_{Lev} \tag{3.5}$$

An extension to the Levenshtein distance that also takes transpositions between two neighbored characters into account is the Damerau-Levenshtein distance (Hyyrö, 2003). The computation of the metric is again solved by a dynamic programming matrix that follows an extended set of rules. Just as for the Levenshtein metric, the first column and row is set to a fixed value $D_{i,0} = i$ and $D_{0,j} = j$ and matching characters require no cost, i.e., $D_{i,j} = D_{i-1,j-1}$ if $a_i = b_j$. The cost for two consecutive characters $a_{i-1}a_i$ or $b_{j-1}b_j$ that are swapped in a or b is only counted as a cost of 1 instead of the cost 2 that would be created by two mismatches. If there is a swap of the characters, i.e., $a_{i-1} = b_j$ and $a_i = b_{j-1}$, and further the previous characters were not swapped already, i.e., $D_{i-1,j-2} > D_{i-2,j-2}$, then the cost for the transposition was already covered by the mismatch of the previous characters and thus $D_{i,j} = D_{i-1,j-1}$. In every other case the same rule $D_{i,j} = 1 + min(D_{i,j-1}, D_{i-1,j}, D_{i-1,j-1})$ that is already known from the Levenshtein method is applied. The normalized distance as well as the normalized similarity are computed identically as it was done with the Levenshtein metric.

3.4 Algorithm

The algorithm used for incrementally generating cluster maps was first published by Wurzenberger et al. (2017) and is in the following explained in detail. Figure 3.2 displays the overall structure of the clustering procedure. At first, the log file is read from hard disk one line after the other and it is assumed that there is a potentially endless amount of incoming lines, i.e., every time a line is printed to the log file it is queued as an input to the algorithm. In the case that there is no new line printed and all the available lines were already processed, the program waits at this point until a new line is available.

Some sample log lines are displayed in Fig. 3.3. As it can be seen, some of the lines show high similarities or are identical while others largely vary in length and content. Computing and utilizing these similarities is the basis for grouping the lines into a cluster. As already mentioned, there are no assumptions made on the structure of the log lines except that there is a time stamp attached that will be used for further time-series analysis. The time stamps can remain unprocessed if it is assumed that the strings are sufficiently long so that the influence on the overall similarity between the lines is small enough to be negligible. However, a better option is to remove the time stamps as they do not convey any information related to the content of the log messages and therefore would only distort the true similarity score between the lines. The same reasoning is applied to characters such as additional spaces and other style-related appendages that should be trimmed from the strings.

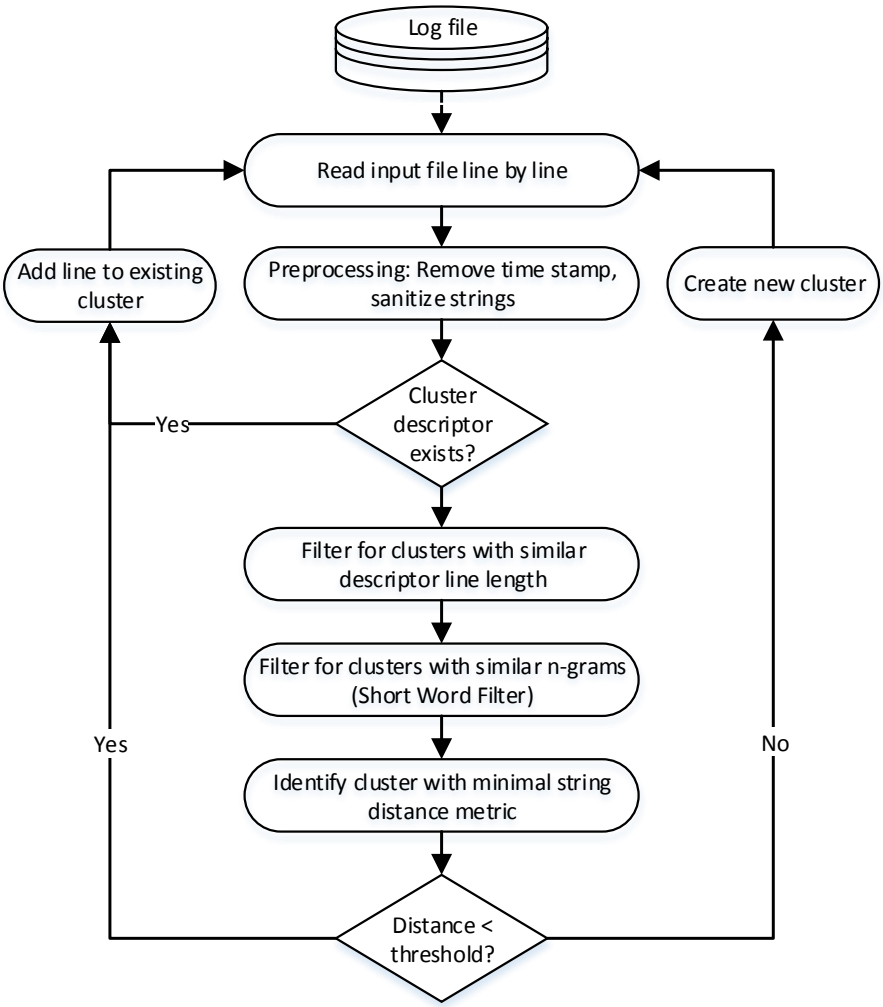

Figure 3.2: Incremental clustering procedure employing a stack of filters for increased performance.

```
Sep 15 11:54:57 reverse-proxy.v3ls1316.d03.arc.local apache: 2079 169.254.0.2:80 "service-3.v3ls1316.d03.arc.local" "service-3.v3ls
Sep 15 11:54:57 service-3.v3ls1316.d03.arc.local apache: 2219 169.254.0.3:80 "mantis-3.v3ls1316.d03.arc.local" "mantis-3.v3ls1316.d
Sep 15 11:54:57 v3ls1316.d03.arc.local mysql-manual-import: 28583 Connect mantis_user_3@service-3.v3ls1316.d03.arc.local on
Sep 15 11:54:57 v3ls1316.d03.arc.local mysql-manual-import: 28583 Init DB mantis_3
Sep 15 11:54:57 v3ls1316.d03.arc.local mysql-manual-import: 28583 Query SET NAMES UTF8
Sep 15 11:54:57 v3ls1316.d03.arc.local mysql-manual-import: 28583 Query SHOW TABLES
Sep 15 11:54:57 v3ls1316.d03.arc.local mysql-manual-import: 28583 Query SELECT basename, priority, protected FROM mantis_plugin_tab
Sep 15 11:54:57 v3ls1316.d03.arc.local mysql-manual-import: 28583 Query SHOW TABLES
Sep 15 11:54:57 v3ls1316.d03.arc.local mysql-manual-import: 28583 Query SELECT config_id, user_id, project_id, type, value, access_
Sep 15 11:54:57 v3ls1316.d03.arc.local mysql-manual-import: 28583 Query SELECT * FROM mantis_user_table WHERE cookie_string='c4bf4b
Sep 15 11:54:57 v3ls1316.d03.arc.local mysql-manual-import: 28583 Query SELECT * FROM mantis_user_pref_table WHERE user_id=4 AND pr
Sep 15 11:54:57 v3ls1316.d03.arc.local mysql-manual-import: 28583 Query DELETE FROM mantis_tokens_table WHERE 1505476497 > expiry
Sep 15 11:54:57 v3ls1316.d03.arc.local mysql-manual-import: 28583 Query SELECT * FROM mantis_tokens_table WHERE type=5 AND owner=4
Sep 15 11:54:57 v3ls1316.d03.arc.local mysql-manual-import: 28583 Query SELECT * FROM mantis_project_table WHERE id=29
Sep 15 11:54:57 v3ls1316.d03.arc.local mysql-manual-import: 28583 Query SELECT * FROM mantis_filters_table WHERE id=18
Sep 15 11:54:57 v3ls1316.d03.arc.local mysql-manual-import: 28583 Query SELECT * FROM mantis_filters_table WHERE user_id=4 AND proj
Sep 15 11:54:57 v3ls1316.d03.arc.local mysql-manual-import: 28583 Query SELECT * FROM mantis_custom_field_table ORDER BY name ASC
Sep 15 11:54:57 v3ls1316.d03.arc.local mysql-manual-import: 28583 Query SELECT cft.id FROM mantis_custom_field_table cft JOIN manti
Sep 15 11:54:57 v3ls1316.d03.arc.local kernel: [1304080.012973] iptables:ACCEPT-INFO IN=eth0 OUT= MAC=08:00:27:f7:94:b2:08:00:27:da
```

Figure 3.3: Excerpt from a log file.

For the next step it is assumed that there is a cluster map $\mathcal{C}$ that holds all clusters $C \in \mathcal{C}$, each of which is represented by the log line that generated it. When the first log line is read from the log file, no cluster exists, i.e., the cluster map is empty. As a consequence, the first line always causes the creation of a new cluster with that line as its cluster representative.

Assuming that at least one cluster exists in the cluster map, every newly incoming line s undergoes the following phases: At first it is checked whether a cluster with a representative r identical to s exists. If such a cluster is found, the line is immediately added to the existing cluster by referencing the line via a unique identifier such as the line number. In case that the number of the line from the text file cannot be determined, a running ID that is incremented by 1 for each new line serves the same purpose. With clusters stored in a hash table, this step can be carried out in $\mathcal{O}(1)$ time. In the case that no fitting cluster was found, the relative differences between the length of the line and the length of each cluster representatives are computed. All clusters where this ratio exceeds a certain similarity threshold $t \in [0, 1]$, i.e., $\frac{min(|s|,|r|)}{max(|s|,|r|)} > t$, are added to a list of cluster candidates $\mathcal{C}_c$. With $|\mathcal{C}|$ being the number of clusters in the map, the time complexity for this step lies in $\mathcal{O}(|\mathcal{C}|)$. As the amount of clusters is growing very fast right at the start of the program but only slowly increasing in the later stages, this cost is considered as more or less constant in long term runs. It is common that occasionally outliers appear, i.e., clusters that only hold very few or only a single line over a long time. If the number of outlier clusters keeps increasing rapidly and checking all cluster representatives becomes too expensive after some time, there is the possibility to remove clusters whose member count does not exceed a certain limit from the cluster map in regular intervals. This is an important step to ensure that the algorithm scales linearly with the number of log lines, as a higher number of lines causes more outliers to appear and thus the computational time would ever increase. It should be noted that this issue will be automatically solved when a new cluster map is generated after each time window has finished. The details about this procedure will be explained in depth in the respective chapters.

In the next phase, the list of cluster candidates is reduced by removing all clusters which representatives are not considered similar to the currently processed log line according to the Short Word Filter criterion that was already explained in detail in the previous section. For simplicity, the same threshold parameter t is used as before. This check is done for every cluster candidate and there is the possibility to enhance the precision of the filter by stacking a combination of N Short Word Filters, each dealing with differently sized n-grams. Clearly, this increases the runtime of this phase, however can reduce the overall runtime in the case that many clusters are successfully eliminated from the candidates list. In practice, the runtime will almost always be faster than the worst case where every exact amount of shared n-grams has to be computed for every representative of $\mathcal{C}_c$ as it is often possible to prematurely determine whether a string will be similar or not, e.g., once one of the N SWFs reports that the required minimum threshold Θ has not been reached it is not necessary to additionally consider the remaining SWFs. Moreover, once Θ of matching n-grams has been reached for one specific n there is no need to continue the computation as it is not necessary to determine the exact amount of matching n-grams. In addition, the n-grams of the representatives of the candidate clusters $\mathcal{C}_c$ are regularly used for comparisons and should therefore be stored in a list rather than computed every time they are needed.

The cluster candidates that remain in $\mathcal{C}_c$ are then used for the final comparison step. The normalized Levenshtein distances between the log line and each of the representatives of the cluster candidates are computed and the cluster with the minimal distance is selected as the best fitting cluster. If the similarity with the representative r of this cluster exceeds the threshold t, i.e., $s_{Lev}(s, r) > t$, the log line is added to the cluster, otherwise a new cluster is created and the log line s is set as the first member and representative of the cluster. The algorithm then jumps back to the start and processes the following log line. Any normalized string distance function can be used for this phase and other metrics may be favorable due to a faster runtime. In general, the Levenshtein distance function has a runtime complexity of $\mathcal{O}(|s| \cdot |r|)$. Due to the length of the strings and the fact that it is likely that the string distance has to be computed multiple times for every incoming log line, this time complexity clearly dominates the whole algorithm. However, the purpose of the prior steps was to reduce the number of cluster candidates in $\mathcal{C}_c$ so that the distance metric has to be computed as few times as possible, thereby saving a tremendous amount of time.

CHAPTER 4

Cluster Evolution

An algorithm for clustering log lines was introduced in the previous chapter. This algorithm is able to operate on a continuous data stream without any fixed end, i.e, the cluster map keeps expanding until the algorithm is manually terminated or until the memory used for storing the references to each line runs out. Several interesting features can be extracted from that procedure, for example, the log line types that are responsible for the largest clusters or outliers that indicate unusual log lines. Unfortunately, most features about the dynamic behaviors of the log line types are lost or difficult to extract from the clusters, although they are highly important for security-relevant analyses. Moreover, all clusters and references to all lines need to be held in memory throughout the run of the algorithm. These problems are solved by generating several smaller cluster maps rather than only one single large map. After a specific amount of time, the features of a cluster map are extracted and then a new empty map is used in the following interval. These periods are called time windows, and it is a non-trivial task to retrieve dynamic information about the clusters from one time window to the other. The aim of cluster evolution is to relate the clusters to each other and to learn about their development.

An illustrative example that gives an overview about the procedure that leads up to the cluster evolution analysis is given in Figure 4.1. In the first step marked with number 1, the incoming log lines are displayed with preceding time stamps. Note that at this point every line is considered equal to all the other lines and the colors and marks were only added for a better visualization of the groupings that will be determined in the clustering step. In this example, three processes ◯, △ and □ produce specific types of log lines, i.e., process ◯ logs user file accesses that appear in random intervals, process △ logs an automated backup procedure that generates lines in regular intervals and □ logs failed login attempts. In step 2, the occurrences of the lines are displayed on a time axis, where t_0, t_1, t_2, t_3 represent the boundaries of the time windows considered for each cluster map. Step 3 then clusters the lines by using any given clustering algorithm and according to any given similarity metric. This results in the three cluster maps $\mathcal{C}, \mathcal{C}', \mathcal{C}''$

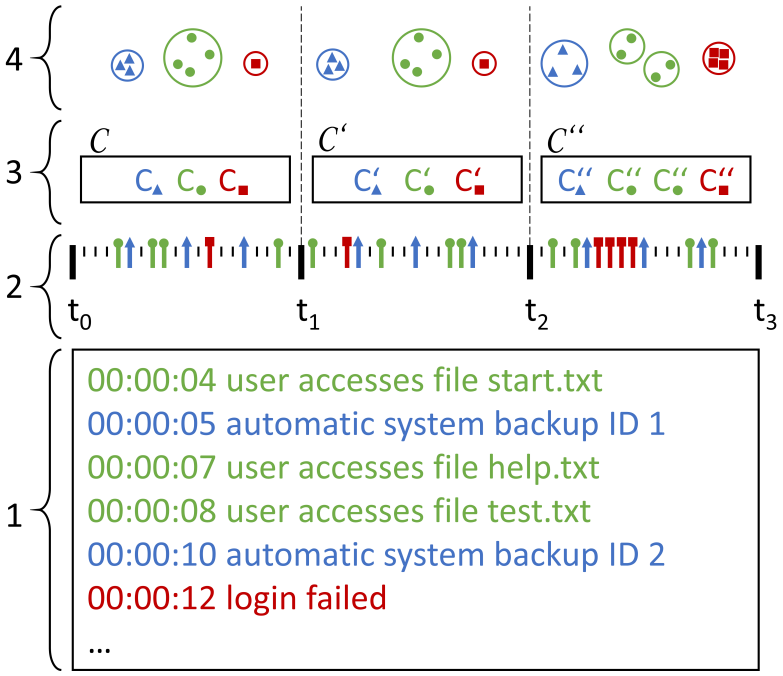

Figure 4.1: Illustrative example of cluster evolution showing a split as well as changes in size, distance and compactness.

from three different time windows. As it can be seen, while in the first two time windows only 3 clusters were found, the cluster map of the final time window consists of 4 clusters. Without cluster evolution it would be hardly possible at this point to determine how the formation of those 4 clusters was accomplished or how any of the clusters from different time windows relate to each other. However, the graphical display of the result of a cluster evolution shown in step 4 gives several useful insights: Not only is it possible to track the clusters over all time steps, it can be seen that cluster $C_{\bigcirc}$ splits up in the last time step. Moreover, cluster $C_{\triangle}$ increased its distance to the other clusters in $\mathcal{C}'$ and further became more diffuse in $\mathcal{C}''$ which is represented by the diameter of the circle. Finally, it can be seen that $C_{\square}$, which represented an outlier in $\mathcal{C}$ and $\mathcal{C}'$, i.e., the line could not be allocated to any other group of messages and thus remained alone in its own cluster, increased its size in $\mathcal{C}''$. All of these effects are indicators for abnormal behavior, for example, the increase of lines that lie inside the 'login failed' cluster may be caused by an attempt to break into a user account by a brute force attack. There are many more possible scenarios for transitions that affect the clusters or long-term trends, all of which are relevant for anomaly detection.

It can therefore be concluded that not only single log lines that do not match any of the existing clusters are of relevance when searching for abnormal system behavior, but also that the properties of the clusters themselves viewed over time are highly important for a thorough anomaly detection. Although it is possible to observe how many samples are

allocated to each cluster when performing the clustering in an incremental fashion, the previously mentioned examples should make clear that information about the evolution of the clusters themselves is still lost. Therefore, there is a need to recluster the data once it is suspected that the clusters do not represent the generated log lines in an appropriate way anymore. Due to restrictions in computational capacity, these reclusterings cannot be made in arbitrarily small time steps (e.g., for every newly arriving log line) and can also not be performed with arbitrarily many data points (e.g., all log lines from the beginning of the recording), but rather need to be started at specific points in time and for a specific time window. This procedure is usually referred to as snapshot analysis where the cluster maps of two distinct points in time are compared to each other, although the snapshot represents a time span during which log lines were generated. Intuitively, the time window can be chosen from the most recent sample to the last sample that was not included in the previous time window. However, it could also be reasonable to include more data points in order to increase the quality of the results, i.e., by allowing an overlap of time windows. It is crucial to determine appropriate values for these parameters based on reasonable metrics in order to continuously ensure a representative clustering of the current behavior of the system while keeping the required computational resources at a minimum.

4.1 Cluster Tracking

Once appropriate values for the required parameters regarding time points and window sizes were chosen and two consecutive cluster maps in the two time windows with their respective sets of clusters $\mathcal{C}$ and $\mathcal{C}'$ are known, it is a non-trivial task to figure out how the set $\mathcal{C}$ transformed into set $\mathcal{C}'$. In order to determine whether a cluster $C \in \mathcal{C}$, for any i, has transformed into cluster $C' \in \mathcal{C}'$, a similarity metric is required. An intuitive approach is to assume that the two clusters C and C' were generated by the same underlying data source if the distance between the majority of the objects contained in C' would have been allocated to cluster C if they had been used for the generation of cluster map $\mathcal{C}$. Greene et al. (2010) employ the following similarity metric that is based on the Jaccard coefficient for binary sets and measures the ratio of common data points in order to determine the overlap:

$$overlap = \frac{|C \cap C'|}{|C \cup C'|} \tag{4.1}$$

They then compare the overlap with a threshold $\theta \in [0, 1]$ to determine whether the clusters match, i.e., whether it can be assumed that C' originates from C. There exist also alternate forms for computing the overlap which use the maximum (Takaffoli et al., 2011) or the minimum (Greene and Cunningham, 2009) of the two set sizes in the denominator and there is also the possibility to use the Hungarian Method, the Max-Flow approach or a linear programming algorithm in order to find the optimal correspondences between clusters of different time steps (Goldberg et al., 2010). In addition, there are variants for

determining whether a transition occurred based on measuring the percentage of change for each cluster (Asur et al., 2009).

It is however problematic to make use of this measure in log file analysis as it is usually a difficult task to determine whether objects that were allocated to two clusters in different time windows are identical, which is a requirement to reasonably perform set operations such as the union and intersection. This is due to the fact that log lines are just strings that can only reliably be tracked by their line number and equality (i.e., an identical sequence of characters) and continuously changing IDs or timestamps contained in the lines could easily cause that highly similar lines from the sets C and C' are not matched due to a single diverging character. The result of this would be that the intersection of the two sets is incorrectly sparse or even empty due to the lack of identical strings.

In order to overcome the previously mentioned problems regarding set operations on log lines, the following strategy was pursued: At first, for reasons regarding memory storage and convenience, each log line is stored and identified by its line number once it is allocated to a cluster. This does not only effectively reduce the amount of required memory as numbers generally require way less storage space than strings, but also ensures that it is precisely known which line is added to the cluster. It is guaranteed that each line is associated with a unique line number, thereby ensuring that the mathematical restrictions regarding identical members in sets is fulfilled. Secondly, the key aspect of this procedure is that the log lines occurring during a certain time window are not only used for creating the cluster map of that time step, but are also allocated to the clusters from the cluster maps preceding and succeeding that map. The two phases are called construction phase and allocation phase respectively. In the construction phase, the cluster maps are generated solely by the log lines that actually occur within that time window. On the other hand, the allocation phase always deals with log lines that are clustered into existing cluster maps from different time windows that lie either before or ahead. For clarification, it should be noted that during the allocation phase the lines do not have any influence on the existing clusters and also do not induce the generation of new clusters in these maps, but are only stored by their line number in the clusters they are allocated to. While a line that does not fall into any existing cluster would form a new one during the construction phase, it is simply omitted if it does not fit into any existing cluster of another map.

An illustrative scenario of this procedure is given in Fig. 4.2. Please note that the colorings in this illustration are again used for an easier differentiation between the clusters and their members and that at the start of the tracking procedure, it is not known that cluster $C'_{\triangle}$ originates from cluster $C_{\triangle}$ and that cluster $C'_{\bigcirc}$ originates from $C_{\bigcirc}$. In this simple example, a log file consisting of 11 lines that are identified with their line numbers is assumed to be the base of the clustering. The log lines $\{s_1, s_2, ..., s_5\}$ are occurring during the first time window and are forming two clusters $C_{\triangle}, C_{\bigcirc} \in \mathcal{C}$. As these are the lines used for the creation of the cluster map in this time window, they are seen as the current members of the respective clusters they belong to. They are therefore stored in the sets of references $R_{\triangle curr} = \{s_1, s_2, s_3\}$ and $R_{\bigcirc curr} = \{s_4, s_5\}$. Analogously

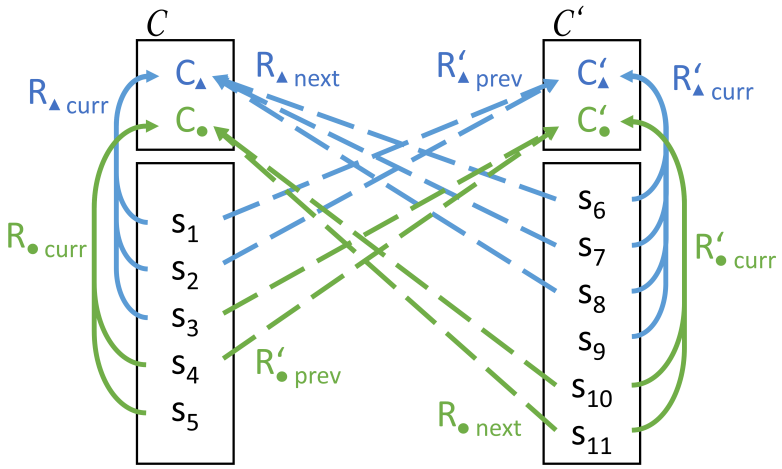

Figure 4.2: Illustrative example how lines are allocated to two different clusters from two consecutive time steps.

for the following time step and the log lines $\{s_6, s_7, ..., s_{11}\}$ occurring during that window, the clusters $C'_{\triangle}, C'_{\bigcirc} \in \mathcal{C}'$ were formed from these lines and thus $R'_{\triangle curr} = \{s_6, s_7, s_8, s_9\}$ and $R'_{\bigcirc curr} = \{s_{10}, s_{11}\}$. The cluster maps are generated and the required references to the generating lines are stored. Thus, the construction phase is finished for this example.

As previously explained, once the two consecutive cluster mappings are established, the allocation phase clusters the lines from each time window into the maps from neighboring time steps. First considering the lines $\{s_1, s_2, ..., s_5\}$ from the former time window being clustered into the map $\mathcal{C}'$ of the later time step, it can be seen that the allocations lead to the sets of references $R'_{\triangle prev} = \{s_1, s_2\}$ and $R'_{\bigcirc prev} = \{s_3, s_4\}$ stored in the clusters. Analogously, the lines from the later time window were allocated to the clusters from the former time step resulting in the references $R_{\triangle next} = \{s_6, s_7, s_8\}$ and $R_{\bigcirc next} = \{s_{10}, s_{11}\}$. It should be noted that line s_3 was clustered into $C_{\triangle}$ in the former time step but into $C'_{\bigcirc}$ in the later time step. A reason for this could be that s_3 has the necessary characteristics to fit into both of the clusters, but due to small deviations in cluster representatives in both time steps the line was not allocated into cluster $C'_{\triangle}$ unlike the lines s_1 and s_2. In a similar manner, the reason why both s_5 and s_9 were not allocated to any cluster from the map of the neighboring time step could be traced to different cluster representatives. These effects are shown on a calculated example using real log lines in Section 4.4.1.

Furthermore, it should be clear that there is an arbitrarily large number of time steps following and that lines have to be clustered accordingly. For example, assuming that there would be a third time step with a cluster map $\mathcal{C}''$ and its clusters $C''_{\triangle}$ and $C''_{\bigcirc}$, their line allocations would be stored in the references $R''_{\triangle curr}$ and $R''_{\bigcirc curr}$. Also, the lines of the second time window would additionally have to be clustered in $\mathcal{C}''$ forming $R''_{\triangle prev}$ and $R''_{\bigcirc prev}$. These connections are not displayed in the figure for simplicity. Finally, the lines of the third time window have to be clustered in $\mathcal{C}'$ forming $R'_{\triangle next}$ and $R'_{\bigcirc next}$. At

the end of the allocation phase, references to all the lines from neighboring time windows are stored in each cluster map.

Using this kind of cluster allocations, the rule for finding matches between clusters stated in Eq. (4.1) is adapted to fit the purpose of log line clustering. Using the line references as explained before, the following formula computes the overlap between any two clusters C and C' of two neighboring maps:

$$overlap(C, C') = \frac{\left|\left(R'_{curr} \cup R'_{prev}\right) \cap (R_{next} \cup R_{curr})\right|}{\left|R'_{curr} \cup R'_{prev} \cup R_{next} \cup R_{curr}\right|} \tag{4.2}$$

Note that by the distributive law and making use of the fact that $R'_{curr} \cap R_{next} = \emptyset$ and $R'_{prev} \cap R_{next} = \emptyset$, this is equivalent to

$$overlap(C, C') = \frac{\left|\left(R_{curr} \cap R'_{prev}\right) \cup (R_{next} \cap R'_{curr})\right|}{\left|R'_{curr} \cup R'_{prev} \cup R_{next} \cup R_{curr}\right|} \tag{4.3}$$

which could allow a more efficient implementation as the intersection is carried out on smaller sets. Furthermore, this representation also shows more clearly that the sets R_{curr} and R'_{prev} both contain log lines that were used in the former time step which was also used to create the cluster map $\mathcal{C}$, while both R_{next} and R'_{curr} contain log lines from cluster map $\mathcal{C}'$, thus showing that the intersections are applied reasonably. Dividing the union of these two intersected sets by the union of all sets means that the resulting value is in the interval $[0, 1]$, with 1 indicating a perfect match (i.e., all lines that were clustered into C were also clustered into C' and vice versa) and 0 indicating a total mismatch.

A more sophisticated clustering model that not only allocates the log lines from a certain time window into the cluster maps of its directly neighboring time windows but also into the ones following after that is able to compute an aggregated overlap over multiple time windows. This means that the overlap from a specific cluster, say $C^1 \in \mathcal{C}^1$, through another cluster $C^2 \in \mathcal{C}^2$ to a third cluster $C^3 \in \mathcal{C}^3$ is computed by not only incorporating the already used references R^1_{next} and R^2_{prev} between C^1 and C^2 as well as R^2_{next} and R^3_{prev} between C^2 and C^3, but also the references between C^1 and C^3. These references are called $R^1_{next,2}$ and $R^3_{prev,2}$, where the additional subscript 2 indicates the distance between the two cluster maps, i.e., cluster map $\mathcal{C}^2$ was skipped. Following this terminology, the references between two directly neighboring cluster maps are called $R^1_{next,1}$, $R^2_{prev,1}$, etc. Analogously, the references between clusters C^i and C^{i+m} that are m steps apart are called $R^i_{next,m}$ and $R^{i+m}_{prev,m}$. The overlap between a sequence of N clusters $C^1, C^2, ..., C^N$ is then defined as

$$overlap(C^1, C^2, ..., C^N) = \frac{\sum_{j=1}^{N-1} \sum_{i=1}^{N-j} \left|\left(R^i_{curr} \cap R^{i+j}_{prev,j}\right) \cup \left(R^i_{next,j} \cap R^{i+j}_{curr}\right)\right|}{\sum_{j=1}^{N-1} \sum_{i=1}^{N-j} \left|R^{i+j}_{curr} \cup R^{i+j}_{prev,j} \cup R^i_{next,j} \cup R^i_{curr}\right|} \tag{4.4}$$

Due to the increased complexity of the sophisticated clustering model, the simple overlap metric is used in the following and thus the additional index specifying the distance between the cluster maps is omitted.

Applying the overlap metric to the illustrative example from Fig. 4.2 allows determining the likelihood that each cluster from the former time step transformed into any other cluster from the later time step. The predefined threshold θ that can arbitrarily be chosen in the range $[0, 1]$ is set to 0.5 for this example. The overlap between clusters $C_\triangle$ and $C'_\triangle$ is computed as

$$\begin{aligned} overlap(C_\triangle, C'_\triangle) &= \frac{|(\{s_6, s_7, s_8, s_9\} \cup \{s_1, s_2\}) \cap (\{s_6, s_7, s_8\} \cup \{s_1, s_2, s_3\})|}{|\{s_6, s_7, s_8, s_9\} \cup \{s_1, s_2\} \cup \{s_6, s_7, s_8\} \cup \{s_1, s_2, s_3\}|} \\ &= \frac{|\{s_1, s_2, s_6, s_7, s_8, s_9\} \cap \{s_1, s_2, s_3, s_6, s_7, s_8\}|}{|\{s_1, s_2, s_3, s_6, s_7, s_8, s_9\}|} \\ &= \frac{|\{s_1, s_2, s_6, s_7, s_8\}|}{|\{s_1, s_2, s_3, s_6, s_7, s_8, s_9\}|} = \frac{5}{7} \approx 0.714 \end{aligned}$$

and thus shows a match between the clusters as $0.714 > \theta$. The overlaps between the remaining combinations of clusters are computed in a similar fashion: The overlap between $C_\bigcirc$ and $C'_\bigcirc$ is 0.6 and is thus also a match, the overlap between $C_\triangle$ and $C'_\bigcirc$ is only 0.111 and thus not a match and the overlap between $C_\bigcirc$ and $C'_\triangle$ results in 0 and is therefore also not a match. According to these results, it can be concluded that cluster $C_\triangle$ transformed into $C'_\triangle$ and cluster $C_\bigcirc$ transformed into $C'_\bigcirc$. However, clusters do not necessarily have to have exactly one predecessor and one successor, but can be the product of multiple clusters that merged together or be a part of a larger cluster that split up. In the following, a method for the identification of these transformations is given.

4.2 Cluster Transitions

The detection of cluster transitions depends on the formulation of the used overlap measure. Existing algorithms therefore cannot be directly applied to the overlap metric defined in Eq. (4.2), but have to be adapted according to its characteristics. Spiliopoulou et al. (2006) employ a related overlap metric that obviously does not make use of the previously explained clustering model and therefore clusters do not contain references to neighboring elements such as R_{next}, but only references to their generating elements. Adapting the notation of the formula to the previously used terminology, their overlap metric is defined as

$$overlap_{Spiliopoulou} = \frac{R_{curr} \cap R'_{curr}}{R_{curr}} \tag{4.5}$$

Using this metric, the authors then propose an algorithm to detect the following important external cluster transitions between the clusters $C \in \mathcal{C}$ and $C' \in \mathcal{C}'$:

1. **Survival**: The cluster C survives and transforms into C' if C' matches C and does not match any other cluster $C_i \in \mathcal{C}$, where $C \neq C_i$.
2. **Split**: The cluster C splits into p multiple clusters $C'_1, C'_2, ..., C'_p$ if each individual C'_j matches C for all j and the union of C'_j matches C for all j and there exists no

other cluster $C'_i \in \mathcal{C}'$ that matches C, with $C \neq C_i$. It should be noted that the separated parts are generally not able to overlap their original cluster and thus a lower threshold $\theta_{part} < \theta$ is used for determining the match of C'_j with C.

3. **Absorption**: The p clusters $C_1, C_2, ..., C_p$ are absorbed by C' if C' matches C_j for all j.

4. **Disappearance**: The cluster C disappears if none of the above cases hold true for C.

5. **Emergence**: A new cluster C' appears if it cannot be matched with any $C_i \in \mathcal{C}$.

Note that the measure used for matching in this algorithm is non-symmetric as it computes the relative shared amount from a cluster in the former time step to another cluster in the later time step, while the overlap measure defined in the previous section is symmetric as it is based on the amount of members that the clusters share in both directions. Even though the threshold for matching is restricted to the range $[0.5, 1]$, a non-symmetric measure allows that the overlap between a cluster from the later time step and multiple clusters from the former time step fulfills the matching criterion. This property is used when identifying absorptions and hence there is no need to compare the overlap with θ_{part} as it is done for the detection of splits.

However, non-symmetric measures do not have this property as the sum of all overlaps between a specific cluster and all clusters from the preceding time window as well as the sum of all overlaps between a specific cluster and all clusters from the succeeding time window never exceeds 1. For this reason, the concept for identifying splits has to be transferred to that of detecting absorptions, i.e., an overlap between two clusters that exceeds θ_{part} is stored as an absorption candidate and finally identified as an absorption if also the overlap between the union of the absorbed cluster candidates and their resulting cluster exceeds θ.

Using the overlap metric as defined in Eq. (4.2), the resulting enumeration of possible cluster transitions and their characteristics is thus as follows:

1. **Survival**: A cluster C survives and transforms into C' if $overlap(C, C') > \theta$ and there exists no other cluster $C_i \in \mathcal{C}$ or $C'_i \in \mathcal{C}'$ so that $overlap(C_i, C') > \theta_{part}$ or $overlap(C, C'_i) > \theta_{part}$.

2. **Split**: A cluster C splits into the parts $C'_1, C'_2, ..., C'_p$ if all individual parts share a minimum amount of similarity with the original cluster, i.e., $overlap(C, C'_j) > \theta_{part}, \forall j$, and the union of all parts matches the original cluster, i.e., $overlap(C, \bigcup C'_j) > \theta$. There must not exist any other cluster that yields an overlap larger than θ_{part} with any of the clusters involved.

3. **Absorption**: The group of clusters $C_1, C_2, ..., C_p$ merge into a larger cluster C' if all individual parts share a minimum amount of similarity with the resulting cluster,

i.e., $overlap(C_j, C') > \theta_{part}, \forall j$, and the union of all parts matches the resulting cluster, i.e., $overlap(\bigcup C_j, C') > \theta$. Again, there must not exist any other cluster that yields an overlap larger than θ_{part} with any of the clusters involved.

4. **Disappearance**: A cluster C disappears if there exists no $C'_i \in \mathcal{C}'$ so that $overlap(C, C'_i) > \theta_{part}$.

5. **Emergence**: A cluster C' emerges if there exists no $C_i \in \mathcal{C}$ so that $overlap(C_i, C') > \theta_{part}$.

The complete procedure for identifying external cluster transitions is shown in pseudo-code in Algorithm 4.1. Line 1 initializes the empty list of all transitions represented as pairs of clusters that are connected over two time steps that will be returned at the end of the algorithm. Line 2 initializes an array that holds a list of all predecessor candidates that can be accessed by a cluster from the later time step, i.e., any C'. Line 3 initializes another array that holds the summed overlaps for those predecessor candidates and can be accessed identically. Analogously, Lines 5-6 initialize those arrays for successors that can be accessed by a cluster from the former time step, i.e., any C. If the overlap between any combination of clusters from different time steps computed in Line 8 exceeds θ_{part} in Line 9, the arrays are updated with this potential connection between the currently processed clusters. In Line 16 it is checked whether the accumulated overlap is larger than θ and only then the connection between a cluster from the former time window and all its successors is added to the list of transitions. Analogously, Line 21 checks the accumulated overlap between a cluster from the later time step and all its predecessors and updates the list of transitions accordingly.

In addition to external transitions, any cluster is subject to undergoing internal transitions regarding one of the following properties:

1. **Size**: The cluster grows in size if C' contains more data points than C, shrinks if C' contains less data points than C and does not change in size otherwise. The size of a cluster C is usually denoted as $|C|$.

2. **Compactness**: With σ denoting the standard deviation of the distance of the cluster members to the representative of cluster C, the cluster becomes more compact if $\sigma' < \sigma$, becomes more diffuse if $\sigma' > \sigma$ and does not change in compactness otherwise.

3. **Location**: Clusters are often defined with coordinates in an arbitrary dimensional space, and therefore it is reasonable to compute the absolute differences between each of the coordinates of C and C' in order to determine whether the cluster moved or remained at the same location. In the field of log line clustering however, distances can only be computed relative to other clusters and this procedure may therefore not be appropriate. However, the average distance to all the other cluster centers may be a reasonable alternative to this metric.

Algorithm 4.1: Determining external cluster transitions between two time steps

```
Data: cluster maps C, C'
Result: list of transitions between pairs of clusters
transitions = List();
predecessorsCandidates = [List()];
predecessorsOverlaps = [ ];
for C ∈ C do
    successorsCandidates = List();
    successorsOverlap = 0.0;
    for C' ∈ C' do
        overlap = computeOverlap(C, C');
        if overlap > θ_part then
            successorsCandidates += C';
            successorsOverlap += overlap;
            predecessorsCandidates[C'] += C;
            predecessorsOverlaps[C'] += overlap;
        end
    end
    if successorsOverlap > θ then
        transitions += {C, successorsCandidates};
    end
end
for C' ∈ C' do
    if predecessorsOverlaps[C'] > θ then
        transitions += {predecessorsCandidates[C'], C'};
    end
end
```

4. **Skewness**: The skewness γ measures the asymmetry between the cluster members and the cluster representative. The skewness of cluster C decreases if $\gamma' < \gamma$, increases if $\gamma' > \gamma$ and remains constant otherwise.

When tracking a cluster through time, it might be of interest to assign some kind of continuous identifier to the evolving cluster in order to easily retrieve the cluster properties in every time window. This identifier is required to be robust to changes in cluster structure due to external and internal transitions. While internal transitions pose less of a problem as the cluster itself remains the same, especially splits and merges make it difficult to allocate such an identifier to the clusters as there should not exist more than one cluster with the same identifier. The most simple solution would be to create a new identifier whenever one of these external transitions occur, however this might not be favorable as it would frequently cut the connection during the dynamic developments

of the clusters resulting in short sequences of values that are retrieved for any evolving cluster.

An exact formulation of a set of rules for allocating identifiers to evolving clusters may depend on the desired outcome of the cluster evolution analysis, the data at hand and the application area. One possibility is based on the assumption that larger cluster sizes indicate more important clusters, while smaller clusters often contain outliers and are of less relevance for anomaly detection. In this setting, new identifiers are only created for clusters that have just emerged or for smaller clusters that separate themselves from existing clusters, with their larger sibling clusters obtaining the identifier from their predecessor. Similarly, the cluster resulting from a merge will retain the identifier from the largest of its predecessors. This makes sure that most of the relevant clusters holding high number of members are tracked successfully, while smaller clusters do not interfere with the overall picture.

Another possibility is the focus on the amount of time windows that the preceding cluster can be tracked back, i.e., its time of existence. It is reasonable to assume that clusters that have already been existing for a longer amount of time are more stable and therefore a better representation of the system. Furthermore, building upon longer existing clusters results in longer time-series that are better fitted for anomaly detection.

Finally, the achieved overlap is another appropriate choice as a higher overlap suggests that the "correct" clusters are connected. This implies that clusters always retain the identifier from the predecessor with the highest overlap and pass the identifier to the successor with the highest overlap. This approach was used when evaluating the experiments carried out in this thesis. In the case that there exist multiple connections that achieve the same overlap score, the identifier could either be transferred randomly or based on another metric. Moreover, a weighted combination of some or all of the previously mentioned metrics could be used to determine the rules for tracking individual clusters.

4.3 Evolution Metrics

It is possible to perform anomaly detection on simple cluster features such as the size. However, the cluster size alone does not always give a complete view about the ongoings of the clusters. For example, a cluster that is the result of a merge does not necessarily change in size, but the fact that a transition is taking place may still be a sign of an anomaly. On the other hand, omitting any information about the advanced transitions may hide that a change of size is caused by a merge. The reaction to such cases may be depending on the application area and the aim of the cluster evolution analysis, but is in any way important to consider in order to understand and capture the interactions between all clusters.

Therefore, measures that represent features of individual clusters, combinations of clusters or the whole cluster map are required. In the following, it is assumed that the sets of

clusters were correctly tracked from one time step to the other so that it is possible to retrieve the references to the log lines that were allocated into each cluster map and for each cluster. A metric that is continuously updated and approximates the current status of the distribution of the cluster members is able to indicate abnormal behavior of members within clusters. However, standard statistical measures like the sample mean and variance cannot be applied directly as they require a fixed and limited set of values rather than a continuous stream. In order to avoid storing all the distances for each line in each cluster until the end of each time window, the online versions of the sample mean and variance introduced in Welford (1962) are used:

1. **Mean distance to cluster representative**: The average distance to the cluster representative is able to provide information about the members of the cluster. For example, the online distance mean remaining 0 means that all lines allocated to this cluster are identical to the representative, while an increase indicates that lines different to the representative have been added to the cluster. Observing the mean thus also provides information about the appropriateness of the chosen distance thresholds, as it should not be the case that all clusters have a mean of 0 due to natural and random noise in the data usually caused by unique IDs and time-dependent variables in the lines. Using any string distance metric $d(r, a)$ that takes the cluster representative r and the currently processed line a as parameters, the sample mean is initialized with $m_1 = d(r, a)$ and updated by

$$m_k = m_{k-1} + \frac{d(r, a) - m_{k-1}}{k}, \tag{4.6}$$

   where k is increased for every line allocated to that cluster. It should be noted that recent values have more influence on the current value of the online mean.

2. **Distance variance**: Statistical measures of the mean usually also require consideration of the variance in order to obtain a proper picture of the distribution of the cluster members. Building upon the equation for the mean, the sample variance is initialized with $s_1^2 = 0$ and continuously updated by

$$T_k = T_{k-1} + (d(r, a) - m_{k-1}) \cdot (d(r, a) - m_k) \tag{4.7}$$

   The variance is then retrieved for any k by $s_k^2 = \frac{T_k}{k-1}$.

While the mean and variance can be measured continuously in each time stamp, there are metrics that take advantage of the fact that the cluster maps are created within a time window and only the final cluster maps retrieved at the end of two consecutive time windows are used for the computation. Toyoda and Kitsuregawa (2003) state several metrics that measure how cluster C changes after its transformation into C'. Note that in the following enumeration the cluster C represents the set of all its contained objects for simplified notation:

1. **Growth rate**: Represents the increase of cluster members per unit time and is able to identify the clusters that are exposed large growing or shrinking effects.

$$R_{growth} = \frac{|C'| - |C|}{t_2 - t_1} \tag{4.8}$$

2. **Stability rate**: Represents the amount of appeared, disappeared, merged and split members per unit time and is thus able to identify clusters that are subject to high changes in population. A stability rate of 0 indicates that the population of the cluster remained the same.

$$R_{stability} = \frac{|C| + |C'| - 2 \cdot |C \cap C'|}{t_2 - t_1} \tag{4.9}$$

3. **Novelty rate**: Represents the amount of newly appeared members per unit time.

$$R_{novelty} = \frac{|C' \setminus C|}{t_2 - t_1} \tag{4.10}$$

4. **Disappearance rate**: Represents the amount of disappeared members per unit time.

$$R_{disappearance} = \frac{|C \setminus C'|}{t_2 - t_1} \tag{4.11}$$

5. **Merge rate**: Represents the amount of members that were absorbed by other clusters per unit time.

$$R_{merge} = \frac{\left|\left(C' \cap \bigcup_{C_i \in \mathcal{C}} C_i\right) \setminus C\right|}{t_2 - t_1} \tag{4.12}$$

6. **Split rate**: Represents the amount of members that were split from C per unit time.

$$R_{split} = \frac{\left|\left(C \cap \bigcup_{C'_i \in \mathcal{C}'} C'_i\right) \setminus C'\right|}{t_2 - t_1} \tag{4.13}$$

A similar issue with set operations on cluster members occurs that was already pointed out for the computation of the overlap, i.e., the assumption that identical objects are clustered in two time windows does not hold true for log lines. Furthermore, all strings need to be kept in memory in order to make this comparison. The growth rate poses an exception as it only requires storing the size of the clusters and can thus always be computed efficiently.

In order to overcome the mentioned issues, the advantages of the bidirectional clustering method that was shown to be successful for computing the overlap are utilized again. In

short, for each cluster map the log lines of the preceding and the succeeding time window have to be clustered into that map and vice versa, as only then the sets of common and different lines can be computed. The metrics given in Eq. (4.8) – (4.13) are therefore adapted and extended for the purpose of log line cluster evolution. A list of these metrics is given in the following:

1. **Absolute growth rate**: Measures the absolute difference between the member sizes of cluster C in two consecutive time steps. Observing this value over time gives a clear overview about growing, shrinking and constant development of the cluster size. It should be clear that resulting growth rates can only be reasonably interpreted if the time windows are of equal length, as a lower or higher amount of lines will be allocated to any cluster if the time window is shortened or prolonged.
$$Growth_{absolute} = |R'_{curr}| - |R_{curr}| \tag{4.14}$$

2. **Relative growth rate**: Measures the relative difference between the member sizes of cluster C in two consecutive time steps with respect to the total number of lines that were clustered in the former time step. Other than the absolute growth, this value takes into account that a lower or higher number of lines occurring in a given time window will also cause the cluster sizes to shrink or grow. This effect is compensated by dividing through the total number of lines from the former time step and is similarly handled for other metrics requiring normalization in the following.
$$Growth_{relative} = \frac{|R'_{curr}| - |R_{curr}|}{\left|\bigcup_{C_i \in \mathcal{C}} R_{curr}\right|} \tag{4.15}$$

3. **Former change rate**: Measures the relative difference between the cluster allocations of the lines from the former time step with respect to the total number of lines that were processed in the corresponding time window. It should be noted that other than the relative growth rate, this metric only takes lines from the former time step into account.
$$Change_{former} = \frac{\left|R'_{prev}\right| - |R_{curr}|}{\left|\bigcup_{C_i \in \mathcal{C}} R_{curr}\right|} \tag{4.16}$$

4. **Later change rate**: Measures the difference between the cluster allocations of the lines from the later time step with respect to the total number of lines that were processed in the corresponding time window. Analogously to the former change rate, it is noteworthy that this metric only takes lines from the later time window into account.
$$Change_{later} = \frac{|R'_{curr}| - |R_{next}|}{\left|\bigcup_{C'_i \in \mathcal{C}'} R'_{curr}\right|} \tag{4.17}$$

5. **Average change rate**: Measures the average relative difference between line allocations from both time steps. Combining both the changes measured on the lines from the former and the later time windows, this metric quantifies the amount of log lines that were allocated to the same cluster but in a different time window. Therefore the change rate is able to express how the role of the cluster changed within the cluster maps, i.e., whether its representative changed.

$$Change_{average} = \frac{Change_{former} + Change_{later}}{2} \tag{4.18}$$

6. **Stability rate**: Measures the fraction of appeared, disappeared, merged and split members between two consecutive time steps. Analogously to previous metrics, this measure is divided into a part only taking the lines from the former time window and another part only taking the lines from the later time window into account. As for all following metrics, both the former and later part are normalized in the range $[0, 1]$ in order to make clusters of different size comparable and to obtain a measure that is comparable with a relative threshold. Note that 0 indicates that all log lines that were allocated to this cluster in one time step were also allocated to this cluster in the other time step and thus the cluster is considered as stable, while 1 indicates that none of the allocations coincided with the other time step and thus the cluster is instable. The average value is again computed in order to incorporate the influences of both time windows into a metric representing the transition from one time step to another.

$$\begin{aligned} Stability_{former} &= \frac{\left|R'_{prev}\right| + |R_{curr}| - 2 \cdot \left|R'_{prev} \cap R_{curr}\right|}{\left|R'_{prev}\right| + |R_{curr}|} \\ Stability_{later} &= \frac{|R'_{curr}| + |R_{next}| - 2 \cdot |R'_{curr} \cap R_{next}|}{|R'_{curr}| + |R_{next}|} \\ Stability_{average} &= \frac{Stability_{former} + Stability_{later}}{2} \end{aligned} \tag{4.19}$$

7. **Novelty rate**: Measures the fraction of newly appeared members between two consecutive time steps. Rather than measuring simple increases in size from one time step to another like the growth rate, the novelty rate focuses on the lines that should have been allocated to a different cluster in the neighboring cluster map but were allocated to this cluster. This also means size changes positively affecting the growth rate do not necessarily lead to a novelty rate different from 0.

$$\begin{aligned} Novelty_{former} &= \frac{\left|R'_{prev} \setminus R_{curr}\right|}{\left|R'_{prev}\right|} \\ Novelty_{later} &= \frac{|R'_{curr} \setminus R_{next}|}{|R'_{curr}|} \\ Novelty_{average} &= \frac{Novelty_{former} + Novelty_{later}}{2} \end{aligned} \tag{4.20}$$

8. **Disappearance rate**: Measures the fraction of disappeared members between two consecutive time steps. Similar to the novelty rate this measure must not be confused with a simple decrease in cluster size, but again focuses on the lines that should have been allocated to this clusters in the neighboring cluster maps but were either allocated to another cluster or no cluster at all.

$$\begin{aligned} Disappearance_{former} &= \frac{\left|R_{curr} \setminus R'_{prev}\right|}{|R_{curr}|} \\ Disappearance_{later} &= \frac{|R_{next} \setminus R'_{curr}|}{|R_{next}|} \\ Disappearance_{average} &= \frac{Disappearance_{former} + Disappearance_{later}}{2} \end{aligned} \tag{4.21}$$

9. **Merge rate**: Measures the fraction of members that were absorbed by other clusters between two consecutive time steps. Note that contrary to the novelty rate, the later set of lines is intersected with the union of the same lines all being clustered in the former cluster map in $Merge_{former}$ and the former set of lines is intersected with the union of the same lines all being clustered in the later cluster map in $Merge_{later}$. Therefore, if all the lines would be successfully allocated to clusters in the maps of the neighboring time steps, then

$$R'_{prev} \subseteq \bigcup_{C_i \in \mathcal{C}} R_{curr}$$

and

$$R'_{curr} \subseteq \bigcup_{C_i \in \mathcal{C}} R_{next}$$

This would lead to the left side of the intersection always being the result of the intersection and thus the merge rate would simplify to the novelty rate. However, if due to changes in cluster representatives some of the log lines cannot be allocated to clusters in the maps of neighboring time steps, those lines would not be contained in the unions and thus be missing from the intersections, thereby providing different information than the novelty rate.

$$\begin{aligned} Merge_{former} &= \frac{\left|\left(R'_{prev} \cap \bigcup_{C_i \in \mathcal{C}} R_{curr}\right) \setminus R_{curr}\right|}{\left|R'_{prev}\right|} \\ Merge_{later} &= \frac{\left|\left(R'_{curr} \cap \bigcup_{C_i \in \mathcal{C}} R_{next}\right) \setminus R_{next}\right|}{|R'_{curr}|} \\ Merge_{average} &= \frac{Merge_{former} + Merge_{later}}{2} \end{aligned} \tag{4.22}$$

10. **Split rate**: Measures the fraction of members that were split from C between two consecutive time steps. The very same phenomenon that was already explained for the merge rate occurs also for the split rate, i.e., successful allocation of all lines into cluster maps of neighboring time steps cause that the split rate degenerates to the disappearance rate. Only lines that are not successfully clustered into these neighboring maps, i.e., lines that are considered outliers from the perspective of the neighboring time step, are causing a different result.

$$
\begin{aligned}
Split_{former} &= \frac{\left| \left(R_{curr} \cap \bigcup_{C'_i \in \mathcal{C}'} R'_{prev} \right) \setminus R'_{prev} \right|}{|R_{curr}|} \\
Split_{later} &= \frac{\left| \left(R_{next} \cap \bigcup_{C'_i \in \mathcal{C}'} R'_{curr} \right) \setminus R'_{curr} \right|}{|R_{next}|} \\
Split_{average} &= \frac{Split_{former} + Split_{later}}{2}
\end{aligned}
\tag{4.23}
$$

It must be noted that in addition to the separation between previous, current and next cluster members, the time difference that was used as a scaling factor in Eq. (4.8) – (4.13) was omitted as it is assumed that time windows are chosen in regular intervals and that the metrics are only computed between two consecutive cluster maps. Moreover, the stated equations only take exactly two clusters from neighboring time windows into account. There is no obvious answer on how to incorporate the fact that each cluster possibly originates from multiple clusters due to absorption or transforms into multiple clusters due to splitting. It may be the most intuitive approach to take any cluster $C' \in \mathcal{C}'$ and create the union of all its p identified directly preceding clusters $C_1, C_2, ..., C_p \in \mathcal{C}$ for computing the metrics. This can be done analogously for splits. Another possibility would be to compute the metrics for any cluster with every of its preceding or succeeding clusters individually, which would however result in a larger amount of metrics. These metrics could then be combined into a single measure by any aggregation method such as averaging.

4.4 Examples

Besides an illustrative example that spanned over two time windows, the previous chapters mostly established a theoretic understanding for the cluster evolution procedure. This section aims at validating the introduced concepts in a more practical way. In the following, the calculations behind some important aspects are shown in a step-by-step demonstration. In addition, a simulation study on synthetic data uses a large-scale data set in order to point out the characteristics of the evolution metrics.

4.4.1 Calculated Example

In order to demonstrate the computation of relevant values in some specific cases, a more complex sample data set was developed. Figure 4.3 shows the created log lines that were designed to form a reasonable amount of clusters. The occurrences of lines span over a period of 6 minutes and a time window of 1 minute was chosen for the following calculations. In the following, the log lines s_0 to s_{74} are referenced by their line ID. The lines are typical Apache log lines that were manually adapted in order to exhibit certain features that are desired for the following demonstrations. There exist three types of log lines that are marked in the following as: ($\bigcirc$) a successful file access over HTTP, ($\triangle$) an error log caused by an authentication failure and ($\square$) an error caused by a malformed host header.

A short look on the data set should immediately show that log lines belonging to the same type result in a high string similarity as they have most of their content in common, except for some minor differences such as IP addresses and filenames. The Levenshtein metric was used for computing the string similarity. For example, log line s_0 achieves a high similarity score of 0.98 to the similar line s_1, however only a low similarity score of 0.15 to the line s_2. The similarity threshold for clustering was set to 0.9 in this demonstration.

Figure 4.4 gives an overview of the cluster maps $\mathcal{C}^1, ..., \mathcal{C}^6$ that were formed in each time window. The elements displayed in each cluster refer to the amount of log lines contained in that cluster. In the first time window that lies within t_0 and t_1, two clusters corresponding to $\triangle$ and $\bigcirc$ are formed by the lines $s_0, ..., s_8$. Cluster $C^1_{\triangle}$ contains $R^1_{\triangle curr} = \{s_2, s_5, s_7, s_8\}$ and Cluster $C^1_{\bigcirc}$ contains $R^1_{\bigcirc curr} = \{s_0, s_1, s_3, s_4, s_6\}$. The first elements contained in the sets of referenced lines of the current time windows are treated as the representatives of these clusters. The next time window establishes a cluster map based on the lines $s_9, ..., s_{17}$. Lines s_9 and s_{10} only achieve a similarity score of 0.88, which is smaller than the predefined minimum similarity threshold. Line s_{10} is therefore not added to the cluster formed by s_9, but rather generates a new cluster. This means that there exist two clusters originating from $\bigcirc$ in this time window. All the clusters from $\mathcal{C}^2$ and their referenced members are therefore: $C^2_{\triangle}$ with $R^2_{\triangle curr} = \{s_{11}, s_{16}, s_{17}\}$, $C^2_{\bigcirc_1}$ with $R^2_{\bigcirc_1 curr} = \{s_{10}, s_{12}, s_{13}, s_{15}\}$ and $C^2_{\bigcirc_2}$ with $R^2_{\bigcirc_2 curr} = \{s_9, s_{14}\}$.

As required by the cluster model, the log lines that generated $\mathcal{C}^2$ are also allocated to the clusters in $\mathcal{C}^1$. This yields the following additional references: Cluster $C^1_{\triangle}$ obtains $R^1_{\triangle next} = \{s_{11}, s_{16}, s_{17}\}$ and $C^1_{\bigcirc}$ obtains $R^1_{\bigcirc next} = \{s_9, s_{10}, s_{12}, s_{13}, s_{15}\}$. Then, the allocation in the other direction is performed, i.e., the log lines from $\mathcal{C}^1$ are allocated into $\mathcal{C}^2$. This yields the following additional references: $C^2_{\triangle}$ obtains $R^2_{\triangle prev} = \{s_2, s_5, s_7, s_8\}$, $C^2_{\bigcirc_1}$ obtains $R^2_{\bigcirc_1 prev} = \{s_0, s_1, s_3, s_4, s_6\}$ and $C^2_{\bigcirc_2}$ obtains $R^2_{\bigcirc_2 prev} = \emptyset$.

With these values it is possible to determine the transitions between clusters from $\mathcal{C}^1$ to $\mathcal{C}^2$. For this, the overlap metric is computed for all possible connections between clusters.

Line ID	Log Line
0	Oct 13 00:00:01 192.168.2.20 - - "GET /image_version_1.png HTTP/1.0" 200 3395
1	Oct 13 00:00:05 192.168.5.33 - - "GET /image_version_1.png HTTP/1.0" 200 3395
2	Oct 13 00:00:14 [error] [client 1.2.3.4] user test: authentication failure for "/~dcid/test1": Password Mismatch
3	Oct 13 00:00:26 192.168.2.20 - - "GET /image_version_2.png HTTP/1.0" 200 3395
4	Oct 13 00:00:33 192.168.2.20 - - "GET /image_version_1.png HTTP/1.0" 200 3395
5	Oct 13 00:00:35 [error] [client 1.2.3.4] user test: authentication failure for "/~dcid/test2": Password Mismatch
6	Oct 13 00:00:41 192.168.2.20 - - "GET /image_version_2.png HTTP/1.0" 200 3395
7	Oct 13 00:00:54 [error] [client 1.2.3.4] user test: authentication failure for "/~dcid/test3": Password Mismatch
8	Oct 13 00:00:56 [error] [client 1.2.3.4] user test: authentication failure for "/~dcid/test2": Password Mismatch
9	Oct 13 00:01:05 192.168.2.20 - - "GET /image_version_2_temp.png HTTP/1.0" 200 3395
10	Oct 13 00:01:12 192.168.5.33 - - "GET /image_version_2.png HTTP/1.0" 200 3395
11	Oct 13 00:01:18 [error] [client 1.2.3.4] user test: authentication failure for "/~dcid/test1": Password Mismatch
12	Oct 13 00:01:22 192.168.2.20 - - "GET /image_version_2.png HTTP/1.0" 200 3395
13	Oct 13 00:01:25 192.168.5.33 - - "GET /image_version_3.png HTTP/1.0" 200 3395
14	Oct 13 00:01:36 192.168.5.20 - - "GET /image_version_2_temp.png HTTP/1.0" 200 3395
15	Oct 13 00:01:44 192.168.2.20 - - "GET /image_version_2.png HTTP/1.0" 200 3395
16	Oct 13 00:01:49 [error] [client 1.2.3.4] user test: authentication failure for "/~dcid/test3": Password Mismatch
17	Oct 13 00:01:55 [error] [client 1.2.3.4] user test: authentication failure for "/~dcid/test3": Password Mismatch
18	Oct 13 00:02:09 192.168.2.20 - - "GET /image_version_2.png HTTP/1.0" 200 3395
19	Oct 13 00:02:11 192.168.2.20 - - "GET /image_version_3.png HTTP/1.0" 200 3395
20	Oct 13 00:02:24 [error] [client 1.2.3.4] user test: authentication failure for "/~dcid/test1": Password Mismatch
21	Oct 13 00:02:26 [error] [client 1.2.3.4] user test: authentication failure for "/~dcid/test1": Password Mismatch
22	Oct 13 00:02:34 192.168.2.20 - - "GET /image_version_3_temp.png HTTP/1.0" 200 3395
23	Oct 13 00:02:46 [error] [client 1.2.3.4] user test: authentication failure for "/~dcid/test3": Password Mismatch
24	Oct 13 00:02:52 192.168.5.33 - - "GET /image_version_2.png HTTP/1.0" 200 3395
25	Oct 13 00:02:57 192.168.5.33 - - "GET /image_version_3.png HTTP/1.0" 200 3395
26	Oct 13 00:02:58 [error] [client 1.2.3.4] user test: authentication failure for "/~dcid/test2": Password Mismatch
27	Oct 13 00:03:09 192.168.2.20 - - "GET /image_version_3.png HTTP/1.0" 200 3395
28	Oct 13 00:03:13 [error] [client 1.2.3.4] user test: authentication failure for "/~dcid/test2": Password Mismatch
29	Oct 13 00:03:14 192.168.2.20 - - "GET /image_version_4.png HTTP/1.0" 200 3395
30	Oct 13 00:03:20 [error] [client 1.2.3.4] user test: authentication failure for "/~dcid/test2": Password Mismatch
31	Oct 13 00:03:24 192.168.2.20 - - "GET /image_version_4.png HTTP/1.0" 200 3395
32	Oct 13 00:03:26 [error] [client 1.2.3.4] Client sent malformed Host header
33	Oct 13 00:03:27 [error] [client 1.2.3.4] user test: authentication failure for "/~dcid/test3": Password Mismatch
34	Oct 13 00:03:29 [error] [client 1.2.3.4] user test: authentication failure for "/~dcid/test1": Password Mismatch
35	Oct 13 00:03:34 [error] [client 1.2.3.4] Client sent malformed Host header
36	Oct 13 00:03:36 [error] [client 1.2.3.4] user test: authentication failure for "/~dcid/test2": Password Mismatch
37	Oct 13 00:03:38 192.168.2.20 - - "GET /image_version_4.png HTTP/1.0" 200 3395
38	Oct 13 00:03:44 192.168.5.33 - - "GET /image_version_4.png HTTP/1.0" 200 3395
39	Oct 13 00:03:46 192.168.2.20 - - "GET /image_version_5.png HTTP/1.0" 200 3395
40	Oct 13 00:03:50 [error] [client 1.2.3.4] user test: authentication failure for "/~dcid/test1": Password Mismatch
41	Oct 13 00:03:52 [error] [client 1.2.3.4] user test: authentication failure for "/~dcid/test1": Password Mismatch
42	Oct 13 00:03:57 [error] [client 1.2.3.4] user test: authentication failure for "/~dcid/test1": Password Mismatch
43	Oct 13 00:04:05 192.168.2.20 - - "GET /image_version_5_new.png HTTP/1.0" 200 3395
44	Oct 13 00:04:07 [error] [client 1.2.3.4] Client sent malformed Host header
45	Oct 13 00:04:10 192.0.0.20 - - "GET /image_version_4.png HTTP/1.0" 200 3395
46	Oct 13 00:04:16 192.168.2.20 - - "GET /image_version_4_new.png HTTP/1.0" 200 3395
47	Oct 13 00:04:19 [error] [client 1.2.3.4] Client sent malformed Host header
48	Oct 13 00:04:23 [error] [client 1.2.3.4] Client sent malformed Host header
49	Oct 13 00:04:27 192.0.0.20 - - "GET /image_version_5.png HTTP/1.0" 200 3395
50	Oct 13 00:04:28 [error] [client 1.2.3.4] Client sent malformed Host header
51	Oct 13 00:04:33 [error] [client 1.2.3.4] user test: authentication failure for "/~dcid/test3": Password Mismatch
52	Oct 13 00:04:46 [error] [client 1.2.3.4] Client sent malformed Host header
53	Oct 13 00:04:48 192.168.2.20 - - "GET /image_version_5_renew.png HTTP/1.0" 200 3395
54	Oct 13 00:04:51 192.0.0.20 - - "GET /image_version_4_copy.png HTTP/1.0" 200 3395
55	Oct 13 00:04:53 192.168.2.20 - - "GET /image_version_5_renew.png HTTP/1.0" 200 3395
56	Oct 13 00:04:54 192.168.2.20 - - "GET /image_version_6_new.png HTTP/1.0" 200 3395
57	Oct 13 00:04:55 192.0.0.20 - - "GET /image_version_5_copy.png HTTP/1.0" 200 3395
58	Oct 13 00:05:03 192.168.2.20 - - "GET /image_version_6_new.png HTTP/1.0" 200 3395
59	Oct 13 00:05:04 127.0.0.20 - - "GET /image_version_6.png HTTP/1.0" 200 3395
60	Oct 13 00:05:06 [error] [client 1.2.3.4] Client sent malformed Host header
61	Oct 13 00:05:09 192.168.2.20 - - "GET /image_version_6_new.png HTTP/1.0" 200 3395
62	Oct 13 00:05:12 127.0.0.20 - - "GET /image_version_6.png HTTP/1.0" 200 3395
63	Oct 13 00:05:15 127.0.0.20 - - "GET /image_version_5.png HTTP/1.0" 200 3395
64	Oct 13 00:05:20 [error] [client 1.2.3.4] Client sent malformed Host header
65	Oct 13 00:05:22 127.0.0.20 - - "GET /image_version_5.png HTTP/1.0" 200 3395
66	Oct 13 00:05:24 192.168.2.20 - - "GET /image_version_6_new.png HTTP/1.0" 200 3395
67	Oct 13 00:05:27 [error] [client 1.2.3.4] Client sent malformed Host header
68	Oct 13 00:05:33 127.0.0.20 - - "GET /image_version_6.png HTTP/1.0" 200 3395
69	Oct 13 00:05:36 127.0.0.20 - - "GET /image_version_5.png HTTP/1.0" 200 3395
70	Oct 13 00:05:39 [error] [client 1.2.3.4] Client sent malformed Host header
71	Oct 13 00:05:42 192.168.2.20 - - "GET /image_version_6_new.png HTTP/1.0" 200 3395
72	Oct 13 00:05:46 [error] [client 1.2.3.4] Client sent malformed Host header
73	Oct 13 00:05:51 192.168.2.20 - - "GET /image_version_6_new.png HTTP/1.0" 200 3395
74	Oct 13 00:05:53 127.0.0.20 - - "GET /image_version_6.png HTTP/1.0" 200 3395

Figure 4.3: Sample log lines used for the demonstration of a calculated example.

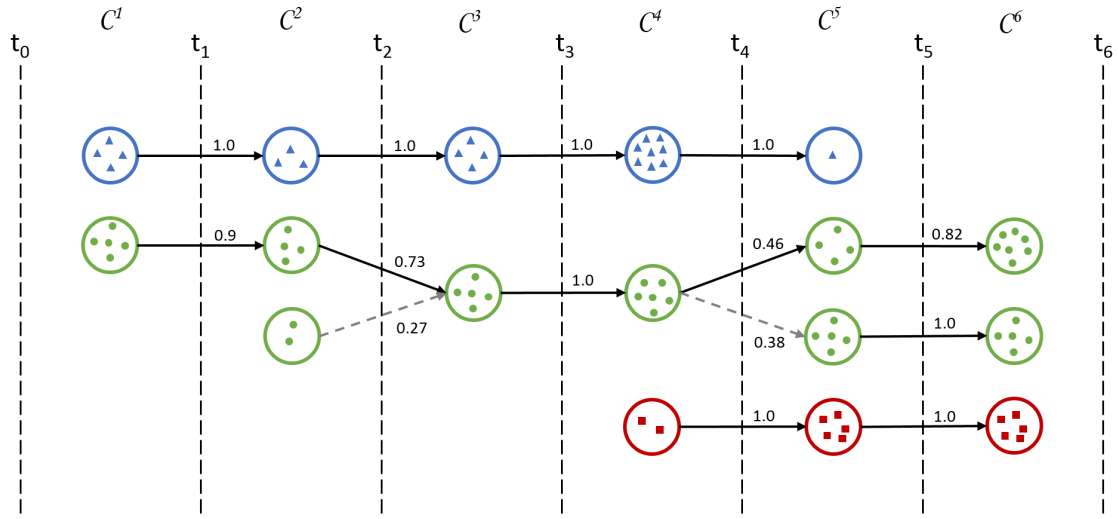

Figure 4.4: Exemplary cluster evolutions over a total of 6 time windows.

The overlap between $C^1_\triangle$ and $C^2_\triangle$ is

$$
\begin{aligned}
overlap(C^1_\triangle, C^2_\triangle) &= \frac{\left|\left(R^1_{\triangle curr} \cap R^2_{\triangle prev}\right) \cup \left(R^1_{\triangle next} \cap R^2_{\triangle curr}\right)\right|}{\left|R^2_{\triangle curr} \cup R^2_{\triangle prev} \cup R^1_{\triangle next} \cup R^1_{\triangle curr}\right|} \qquad (4.24)\\
&= \frac{|(\{s_2, s_5, s_7, s_8\} \cap \{s_2, s_5, s_7, s_8\}) \cup (\{s_{16}, s_{17}, s_{11}\} \cap \{s_{16}, s_{17}, s_{11}\})|}{|\{s_{16}, s_{17}, s_{11}\} \cup \{s_2, s_5, s_7, s_8\} \cup \{s_{16}, s_{17}, s_{11}\} \cup \{s_2, s_5, s_7, s_8\}|}\\
&= \frac{|\{s_2, s_5, s_7, s_8\} \cup \{s_{16}, s_{17}, s_{11}\}|}{|\{s_2, s_5, s_7, s_8, s_{16}, s_{17}, s_{11}\}|}\\
&= \frac{|\{s_2, s_5, s_7, s_8, s_{16}, s_{17}, s_{11}\}|}{|\{s_2, s_5, s_7, s_8, s_{16}, s_{17}, s_{11}\}|} = \frac{8}{8} = 1
\end{aligned}
$$

According to the transition detection algorithm, the overlap between two clusters must be at least θ_{part} in order to be considered a candidate for a split or merge and the overall overlap must be at least θ in order to be added to the set of transitions. In this demonstration, the thresholds are set to $\theta_{part} = 0.2$ and $\theta = 0.7$. The clusters $C^1_\triangle$ and $C^2_\triangle$ reach the highest possible overlap score of 1, i.e., all log lines that were clustered into $C^1_\triangle$ were allocated to $C^2_\triangle$ and vice versa. Therefore, the link between these two clusters is immediately established.

All the other combinations of clusters from two neighboring cluster maps are computed

analogously. Two clusters that do not yield a perfect overlap score are $C^1_{\bigcirc}$ and $C^2_{\bigcirc_1}$:

$$overlap(C^1_{\bigcirc}, C^2_{\bigcirc_1}) = \frac{\left|\left(R^1_{\bigcirc curr} \cap R^2_{\bigcirc_1 prev}\right) \cup \left(R^1_{\bigcirc next} \cap R^2_{\bigcirc_1 curr}\right)\right|}{\left|R^2_{\bigcirc_1 curr} \cup R^2_{\bigcirc_1 prev} \cup R^1_{\bigcirc next} \cup R^1_{\bigcirc curr}\right|} \tag{4.25}$$

$$= \frac{|(\{s_0, s_1, s_3, s_4, s_6\} \cap \{s_0, s_1, s_3, s_4, s_6\}) \cup (\{s_9, s_{10}, s_{12}, s_{13}, s_{15}\} \cap \{s_{10}, s_{12}, s_{13}, s_{15}\})|}{|\{s_{10}, s_{12}, s_{13}, s_{15}\} \cup \{s_0, s_1, s_3, s_4, s_6\} \cup \{s_9, s_{10}, s_{12}, s_{13}, s_{15}\} \cup \{s_{10}, s_{12}, s_{13}, s_{15}\}|}$$

$$= \frac{|\{s_0, s_1, s_3, s_4, s_6\} \cup \{s_{10}, s_{12}, s_{13}, s_{15}\}|}{|\{s_0, s_1, s_3, s_4, s_6, s_9, s_{10}, s_{12}, s_{13}, s_{15}\}|}$$

$$= \frac{|\{s_0, s_1, s_3, s_4, s_6, s_{10}, s_{12}, s_{13}, s_{15}\}|}{|\{s_0, s_1, s_3, s_4, s_6, s_9, s_{10}, s_{12}, s_{13}, s_{15}\}|} = \frac{9}{10} = 0.9$$

Again this overlap is high enough so that the connection between $C^1_{\bigcirc}$ and $C^2_{\bigcirc_1}$ is established immediately. The reason why no perfect overlap of 1 was achieved in this case is that line s_9 was clustered into $C^2_{\bigcirc_2}$ rather than $C^2_{\bigcirc_1}$ and is thereby missing in the intersection. Finally, the last relevant overlap between $C^1_{\bigcirc}$ and $C^2_{\bigcirc_2}$ is

$$overlap(C^1_{\bigcirc}, C^2_{\bigcirc_2}) = \frac{|\{s_9\}|}{|\{s_0, s_1, s_3, s_4, s_6, s_9, s_{10}, s_{12}, s_{13}, s_{14}, s_{15}\}|} = \frac{1}{11} \approx 0.09 \tag{4.26}$$

and therefore not high enough to be considered as a candidate for a transition as it does not exceed θ_{part}. The reason for this result is twofold. Firstly, line s_{14} contained in cluster $C^2_{\bigcirc_2}$ was not allocated into $R^1_{\bigcirc next}$ due to a too low similarity to the clusters representative and thus formed an outlier, i.e., was not allocated to any cluster. Secondly, none of the log lines from cluster $C^1_{\bigcirc}$ were allocated to $R^2_{\bigcirc_2 prev}$. All the remaining combinations of clusters result in overlap scores of 0.

The same calculations are carried out between cluster maps $\mathcal{C}^2$ and $\mathcal{C}^3$. While the overlap between $C^2_{\triangle}$ and $C^3_{\triangle}$ is again 1, the transitions between $C^2_{\bigcirc_1}$, $C^2_{\bigcirc_2}$ and $C^3_{\bigcirc}$ form a merge. At first, $overlap(C^2_{\bigcirc_1}, C^3_{\bigcirc}) \approx 0.73 > \theta$ establishes a connection between these two clusters. However, $overlap(C^2_{\bigcirc_2}, C^3_{\bigcirc}) \approx 0.27 > \theta_{part}$ suggests that cluster $C^2_{\bigcirc_2}$ contributes to the resulting cluster $C^3_{\bigcirc}$ and is therefore also added as an additional transition.

It was already mentioned that splits and merges pose a problem for tracking individual clusters. By applying the rule that the highest overlap should be followed, the development of cluster $C^3_{\bigcirc}$ refers to cluster $C^2_{\bigcirc_1}$ rather than $C^2_{\bigcirc_2}$ when tracking log line type $\bigcirc$. For this reason, the connections with lower overlaps are displayed as dashed lines in Fig. 4.4.

Besides the appearance of a new cluster $C^4_{\square}$ and the disappearance of $C^5_{\triangle}$ in the following time window, one more important event occurs between $\mathcal{C}^4$ and $\mathcal{C}^5$: Cluster $C^4_{\bigcirc}$ splits into the two clusters $C^5_{\bigcirc_1}$ and $C^5_{\bigcirc_2}$. Different to the overlaps of the previously discussed merge where one overlap exceeded θ, both $overlap(C^4_{\bigcirc}, C^5_{\bigcirc_1}) \approx 0.46 < \theta$ and $overlap(C^4_{\bigcirc}, C^5_{\bigcirc_2}) \approx 0.38 < \theta$. This means that none of the overlaps alone would be

Figure 4.5: Cluster sizes plotted as time-series. Blue: △, Green: ◯, Red: □.

sufficiently high to establish a connection. However, both overlaps exceed θ_{part} and are therefore handled as candidates for the split. Furthermore, the sum of the overlaps exceeds θ and both connections are therefore added to the set of transitions. Again, the path with the higher overlap (i.e., the link between $C^4_{\bigcirc}$ and $C^5_{\bigcirc_1}$) is followed when tracking the cluster corresponding to ◯.

All the clusters that exist in at least 3 consecutive time windows were tracked. Their sizes are retrieved after every time window and plotted in Figure 4.5. Cluster sizes of 0 are not displayed in the plot as this means that the corresponding clusters do not exist in the respective time windows.

This demonstrative scenario is also appropriate for computing evolution metrics. For example, the relative growth rate of cluster △ between time steps 3 and 4 is calculated by

$$\begin{aligned} Growth_{relative} &= \frac{\left|R^4_{\triangle curr}\right| - \left|R^3_{\triangle curr}\right|}{\left|\bigcup_{C_i \in \mathcal{C}^3} R_{C_i curr}\right|} \\ &= \frac{|\{s_{33}, s_{34}, s_{36}, s_{40}, s_{41}, s_{42}, s_{28}, s_{30}\}| - |\{s_{20}, s_{21}, s_{23}, s_{26}\}|}{|\{s_{18}, s_{19}, s_{20}, s_{21}, s_{22}, s_{23}, s_{24}, s_{25}, s_{26}\}|} \\ &= \frac{8-4}{9} = \frac{4}{9} \approx 0.44 \end{aligned} \tag{4.27}$$

This indicates a rather high growth that can also be seen in Fig. 4.5. Other rates and metrics are calculated analogously.

4.4.2 Simulation Study

The techniques and metrics explained in the previous chapters are complemented with the results of an exemplary simulation that should both ascertain the validity and proper functioning of the introduced approach and also improve the understanding of the abilities and properties of the measures. For the experiment, synthetic log data that contains known anomalies has been created. For simplicity, each type of log line consists only of a sequence of identical characters, e.g., "aaaa...", "bbbb...", etc. In order to incorporate a certain degree of randomness that increases the difficulty for the clustering algorithm to successfully group similar lines together and also approximates real-world data more appropriately, random noise is added to the lines. This is done by occasionally replacing some characters in the strings and randomly altering the length of the lines. In general however, the randomness factor is set to a value that allows the clustering algorithm to allocate most of the lines that are produced by a certain process correctly, otherwise no reasonable evaluation is possible and no conclusions regarding the evolution metrics can be drawn. The processes generating each type of lines were programmed with different behaviors that change over time, including changes in the printing rate or randomness of log line content. In the following, the processes are denoted by the type of log lines they produce, e.g., the process that creates the line "aaaa..." is named "A", etc. The following processes will be investigated in detail:

- "A": The rate in which log lines are produced increases and decreases over time, thereby creating a periodic behavior. The changes are short-term and one period consists of 24 hours, thus simulating a daily task that is frequently occurring in many real-world scenarios.
- "B": The rate is progressing according to a sinusoidal curve. Other than the short-term intervals of process "A", the period of the sinusoidal curve spans over multiple days and is thus simulating an automated task that is not bound to a daily schedule but rather continuously operates in its own and independent interval.
- "C": The rate in which log lines are produced decreases in steps. Eventually, no more log lines of this type are produced at all. This simulates the disappearance of a cluster.
- "D": This process does not produce any log lines at the beginning. Only after some time, the printing rate is increased stepwise from 0. This simulates the emergence of a new cluster and an increase in size.
- "E": No changes, but is affected by process "F".
- "F": At first, log lines are identical to the log lines produced by process "E", i.e., they consist only of the strings "eeee..." and it is not possible to differentiate whether process "E" or "F" produced these lines. However, over time an increasing amount of characters in the lines produced by process "F" are replaced with the

character “f”, so that eventually process “F” only produces the strings “ffff...”. This behavior simulates process “F” splitting up from process “E”.

- “G”: No changes, but affected by process “H”.
- “H”: At first, this process generates the lines “hhhh...” as it could be expected. After some time, an increasing amount of characters is replaced by the character “g”, until eventually process “H” produces identical log lines as process “G”, i.e., both processes produce only “gggg...” and it is not possible to trace from which process the line originates from by analyzing the log file. This simulates process “H” merging into process “G”.
- “I”: The random noise regarding the content of the lines produced by this process is increased in steps. This simulates an increase in spread of cluster members.

It should be noted that other processes exist, but do not exhibit any additional characteristics other than the ones mentioned above and are solely used for a more realistic scenario with a larger amount of clusters, a higher degree of randomness and an increased likelihood of occasional misclassifications.

The data was simulated to cover a period of 2 weeks and was clustered on an hourly basis as explained in Section 3. This means that 336 cluster maps were generated in total. Following the procedure elucidated in this chapter, each log line was not only clustered into the maps in which they occurred, but also in the preceding and succeeding map, except for the lines from the first and last time step where only one neighboring cluster map existed. Once the complete graph representing the dynamic development and dependencies of each cluster was computed, it is possible to trace single clusters and display their properties as a function of time. Figure 4.6 shows the amount of elements allocated into the cluster containing log lines originated by process “A”. Although it is possible that also another process contributes to that cluster due to the content diversity inherent to the generation of log lines, it can be assumed that in this exemplary simulation with a controlled amount of randomness almost all log lines from this cluster originate from process “A”. As it can be seen, the generation of log lines follows a short-term cycle of 24 hours which manifests itself in the plot of the cluster size. On the right-hand side, the absolute growth rate in each time step is displayed. While it is difficult to make out any patterns from this plot, it is interesting to notice that there is a spike at the beginning indicating a large growth and corresponding to the creation of the cluster and that the remaining values are centered around 0. This means that the amount of added and removed elements remains more or less the same and thus showing that there is no constant long-term trend of the cluster size.

The cluster size plot of process “B” that can be seen at the left-hand side in Fig. 4.7 again shows a periodic behavior with a lower frequency than process “A”. Furthermore the plot of the relative growth rate of this cluster is displayed. As the total amount of produced lines of all processes remained almost constant, the relative growth rate

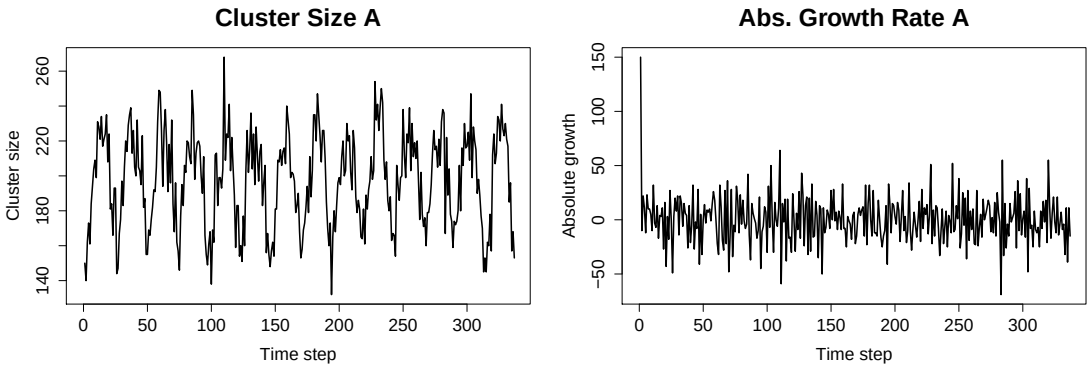

Figure 4.6: Cluster size and absolute growth rate over time of log lines produced by short-term periodic process “A”.

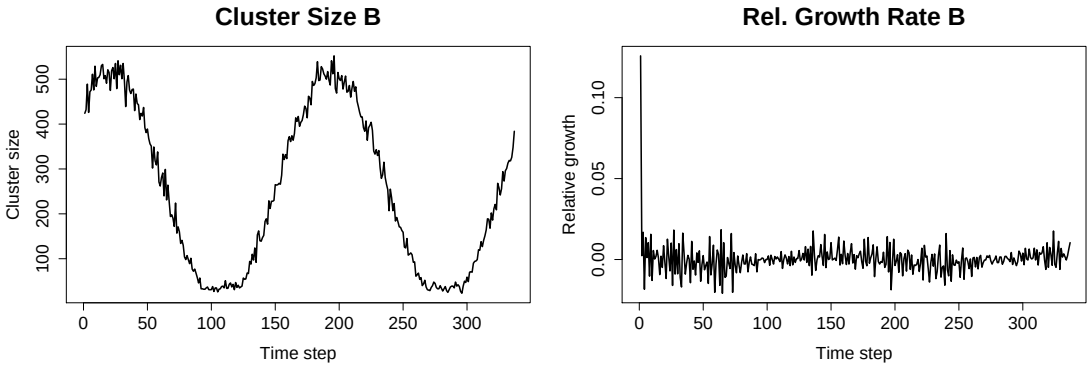

Figure 4.7: Cluster size and relative growth rate over time of log lines produced by long-term periodic process “B”.

shows the same features as the absolute growth rate, only differing in the scale of the y-axis. Due to the low frequency of the curve, the growth rate shows a slight wave-like structure that corresponds to the intervals where the size is increasing or decreasing. Moreover, when the cluster size is at its lowest point, there is less divergence in growth which corresponds to the fewer fluctuations of the growth rate during that phase.

The plots of process “C” can be seen in Fig. 4.8. The size decreases stepwise and the cluster completely disappears eventually. Again the absolute growth rate is shown in the plot on the right-hand side. While the plot is again centered around 0, some downward spikes can be observed that correspond to the loss of elements from the left-hand plot.

Contrary to the decrease in size shown in the previous plot, Fig. 4.9 shows the emergence of a new cluster and its increase in size over time. The relative growth rate correctly shows spikes indicating these events, where the first spike corresponds to the creation and initial filling of the cluster. It should be noted again that the absolute growth rate

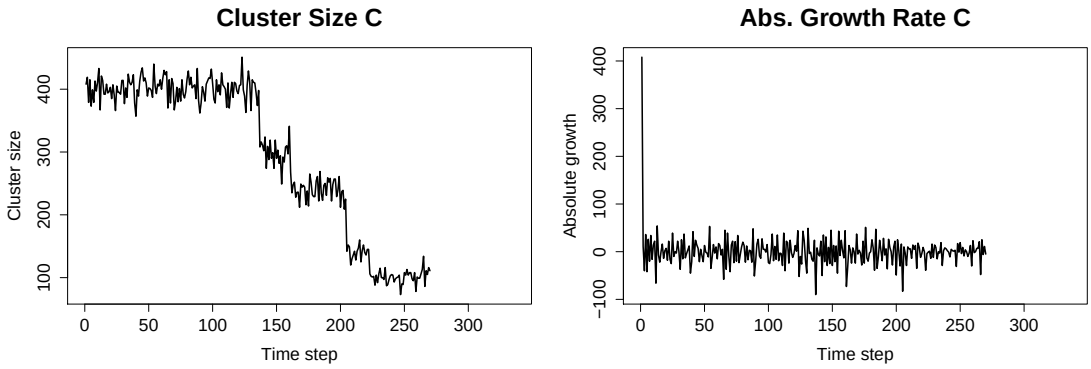

Figure 4.8: Cluster size and absolute growth rate over time of stepwise decreasing log lines produced by process “C”.

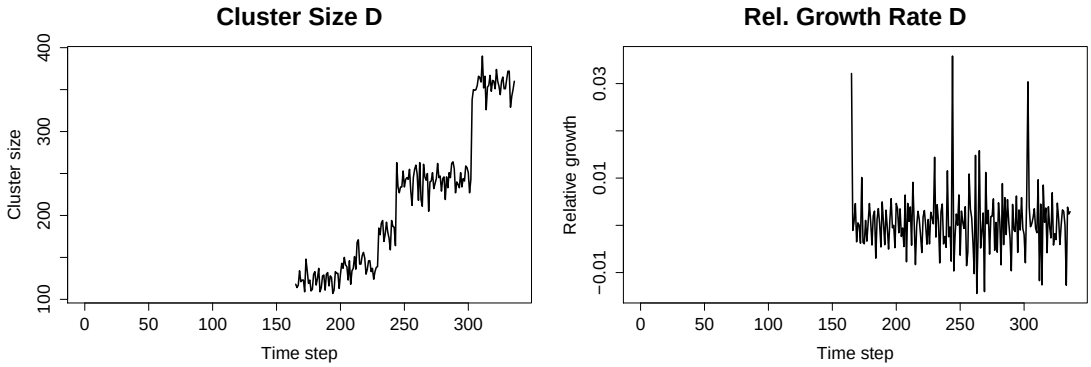

Figure 4.9: Cluster size and relative growth rate over time of stepwise increasing log lines produced by process “D”.

does not show any differences in the overall structure due to the more or less constant total amount of log lines being produced.

For all of the previously shown plots, none of the metrics regarding changes, stability, novelties, disappearances, merges or splits were displayed as they do not show any interesting features and would only deviate from 0 due to random misallocations. This is due to the fact that only the rates in which log lines are printed were changed, however the content of the lines mostly remained the same and thus the clustering algorithm almost always allocated the lines to the same cluster in different time steps. In the case displayed in Fig. 4.10 however, a common cluster containing lines from processes “E” and “F” split apart and thus a large amount of misallocations occur due to the high similarity of cluster representatives in the critical splitting phase. In the top-left plot, the developments of the cluster sizes of log lines created by processes “E” and “F” are shown. As it can be seen, around time step 75 the cluster “E” randomly loses and gains high

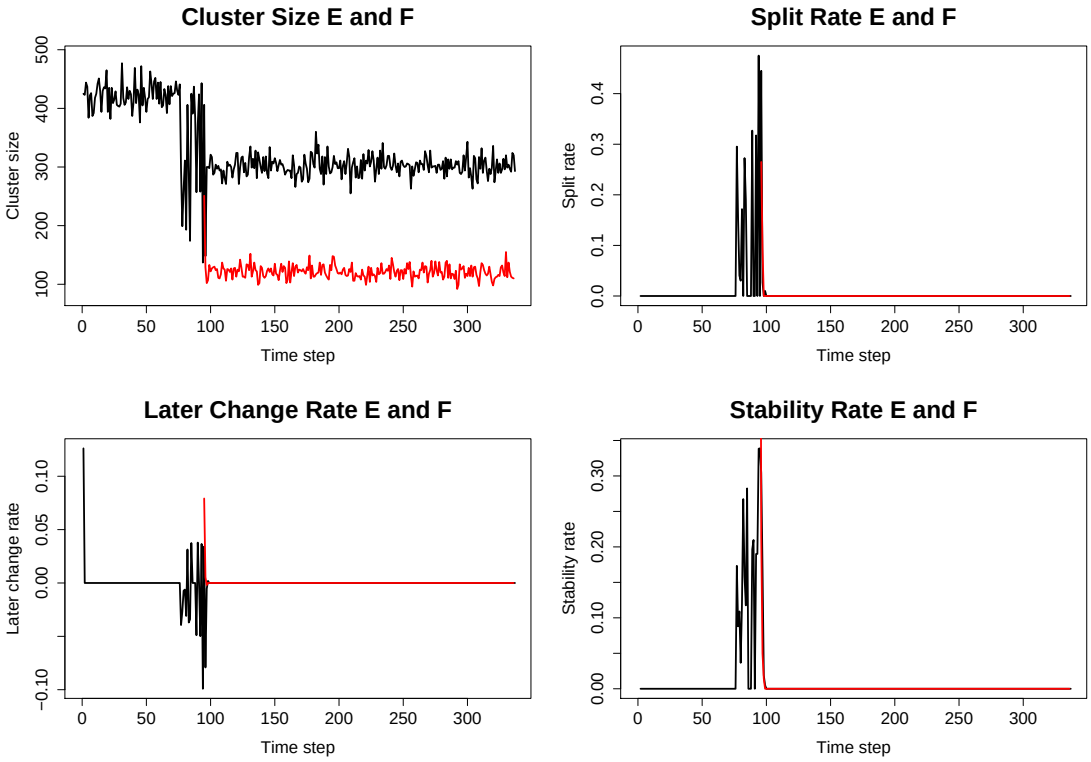

Figure 4.10: Cluster size, split rate, current change rate and stability rate over time of log lines produced by process “F” (red line) splitting from process “E” (black line).

number of elements in every step due to the representatives and log lines only sometimes matching each other so that the strings are correctly allocated to the cluster, while others that do not show enough similarity to the cluster representative are either outliers or form their own cluster that only exists for one or very few time steps. Only at the end of the critical splitting phase, at the point where process “F” regularly produces log lines that are different enough so that no more random misallocations occur, cluster “E” stabilizes again. Moreover, cluster “F” emerges at the same point and continues to exist for the remaining time of the simulation. The other plots of Fig. 4.10 show the developments of the split rate, the later change rate and the stability rate for both process “E” and “F”. All of them show that the clusters undergo a structural change caused by the frequent exchange of elements with other clusters during the splitting phase. It should be noted that also the merge rate would react to the splitting as all the log lines that could not be allocated to cluster “E” in one time step may be correctly allocated in the following step, thus corresponding to the merge of two clusters.

Figure 4.11 shows the merge of clusters “G” and “H”. The top-left plot again shows the evolution of the cluster sizes, with cluster “H” spontaneously disappearing as soon as it is starting to merge with cluster “G”. The merge causes the rapid increase in size of

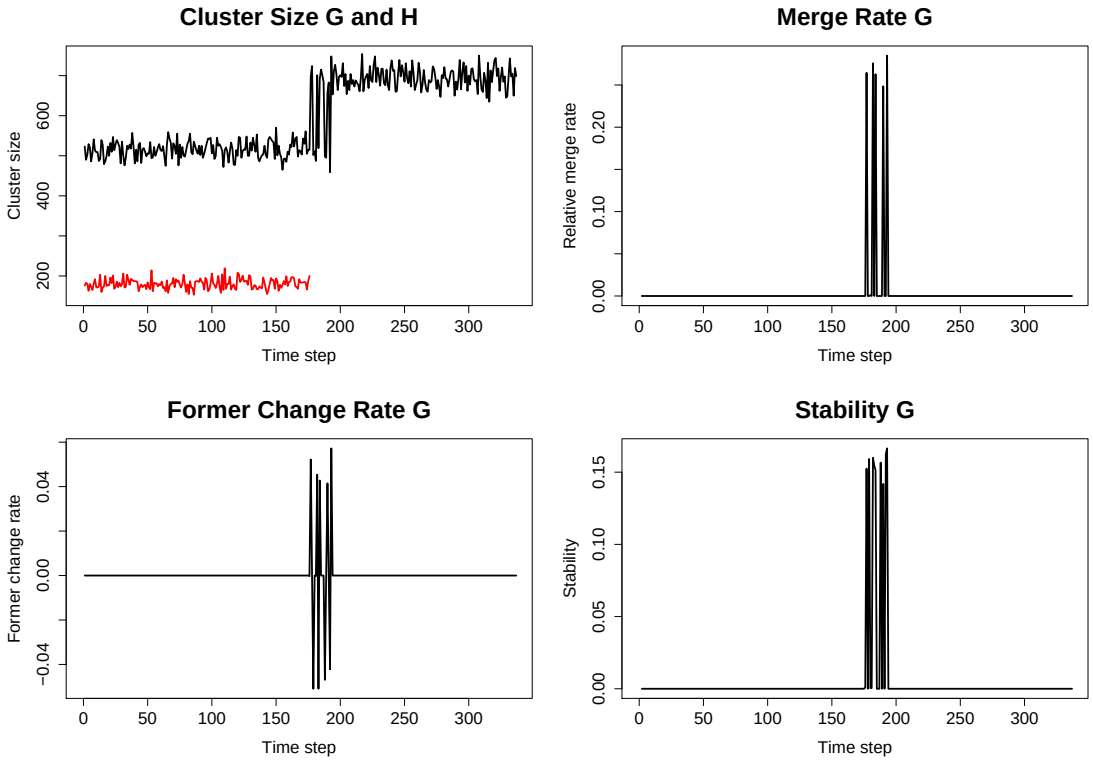

Figure 4.11: Cluster size, merge rate, previous change rate and stability rate over time of log lines produced by process “H” (red line) merging into process “G” (black line).

cluster “G” at time step 175. As for the split, the merging phase takes several time steps until cluster “G” stabilizes and now contains both log lines from processes “G” and “H”. This is due to the fact that the content of the lines was changed stepwise and in other simulations or in real-world applications the merge could occur in a single step. The remaining plots show the merge rate, the former change rate and the stability. The curves appear similar to the curves created by the split as they also indicate the structural changes of the clusters. Judging from the plots it is thus difficult to determine whether a split or merge occurred and it is therefore usually necessary to consult the output of the transition detection algorithm in order to extract the information which clusters contributed to a split or merge.

Finally, Fig. 4.12 shows the effects of a cluster increasing in spread. The plot on the left hand side shows the size of cluster “I” remaining more or less constant until time step 200. Then, many of the log lines generated by process “I” are too dissimilar in order to be allocated to this evolving cluster in different time steps, thus causing the sudden drop of cluster size. Only a few time steps later, the algorithm detecting the cluster transitions was not able to further track this cluster as the lines were too different to form a common cluster and thus the cluster disappears. While there was no way to observe this behavior

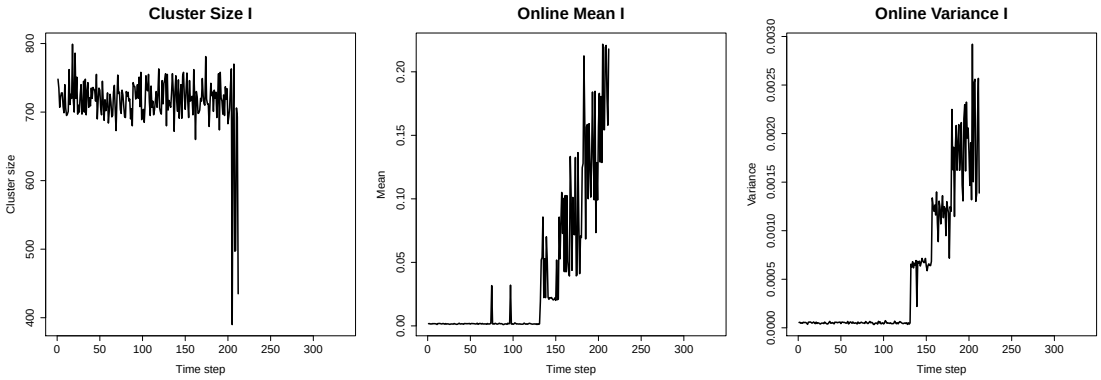

Figure 4.12: Cluster size, mean and variance over time of log lines with stepwise increasing spread produced by process “I”.

in the size plot before time step 200, both the mean and variance of the cluster members that can be seen in the other two plots give a clear view of the stepwise increase of the noise affecting the log lines. As there is no change in the allocation of cluster members, the other rates that were previously considered do not show any interesting features.

CHAPTER 5

Time-series Analysis

Time-series analysis is not only a highly popular and well-researched topic, but also has widespread application areas, among them being finance, econometrics and environment-related fields. This is due to the fact that almost all of the data samples gathered in these areas obtain an inherent dynamic property, i.e., each measurement or value is associated with a specific point in time. The result of a feature being measured in intervals is a sequence of values that is called a time-series. The reasons why time-series analysis is employed are generally grouped into two use cases: First, an improved understanding of historical values, i.e., the creation of models that support reasoning over specific developments of the time-series including trends, fluctuations and spikes. Second, the generation of forecasts of future values based on knowledge gained from the models. In the most cases, forecasting requires a model that is extrapolated beyond its most recent value. In the following, the generation of such a model is explained in detail.

5.1 Models

A time-series is a sequence of N values $y_1, y_2, ..., y_N$ where each of the y_i was recorded at a specific point in time. For convenience it is usually assumed that the interval length between these time points is constant, i.e., that the measurements were taken with a certain frequency. The resulting series of values exhibits numerous characteristics describing the nature of the data. In order to obtain a better understanding of the time-series, one of the first steps is usually to fit a model that approximates the data well and to base all further interpretations, comparisons and predictions on this model.

A simple approach to fit a model is to assume that each value in the time-series is depending on every other preceding point which is known as an autoregressive (AR) process (Cryer and Chan, 2008). As it can be assumed that recent values have more influence on the currently observed point and in order to keep the amount of terms within a reasonable range, only the preceding p points are considered. This parameter is

also called the order of the model and an AR model of order p is denoted as AR(p). In mathematical terms, the dependency of point y_t at time step t can thus be expressed as follows:

$$y_t = a + \phi_1 y_{t-1} + \phi_2 y_{t-2} + ... + \phi_p y_{t-p} + e_t, \tag{5.1}$$

where a is the intercept term, ϕ_i is a coefficient that displays a weight for each term that lags i steps behind and e_t is the error term that includes everything that cannot be explained by the combination of previous values. The error is usually assumed to be independent and equally distributed around a mean of 0.

Another way of describing a time series is by assuming that every value can be displayed as a combination of a mean μ of all error values that occurred up to this point, i.e.,

$$y_t = \mu + e_t - \theta_1 e_{t-1} - \theta_2 e_{t-2} - ... - \theta_q e_{t-q}, \tag{5.2}$$

where it is common to use negative signs in front of the weights θ_i. This kind of model is called a moving-average (MA) process. Similar to the AR process, the terms are limited to q lags behind the current data point and analogously the notation of a MA model order q is MA(q).

For a given time-series $y_1, y_2, ..., y_N$ of length N that is subject to approximation by either an AR or MA process, it is not obvious which model is superior over the other solely based on the data. In order to be able to determine which approach is a better choice for modeling the time-series, an additional measure should be consulted. Recalling that the AR process assumed that the value of each data point is depending on every preceding point, the correlation between the time-series and itself being lagged k steps behind, i.e., the aggregated correlation between the pairs of values $(y_N, y_{N-k}), (y_{N-1}, y_{N-k-1}), ..., (y_k, y_1)$, is used as a measure to describe this dependency. Assuming stationarity, i.e., a constant mean $\bar{y}$ and constant variance over time, the so-called autocorrelation function (ACF) that lies within the range $[-1, 1]$ is defined for any lag $k \geq 0$ as

$$ACF_k = \frac{\sum_{t=k+1}^{N} (y_t - \bar{y}) \cdot (y_{t-k} - \bar{y})}{\sum_{t=1}^{N} (y_t - \bar{y})^2} \tag{5.3}$$

An autocorrelation of 1 indicates that the time-series correlates perfectly with the lagged version of itself, while an autocorrelation of -1 indicates a perfect negative correlation and an autocorrelation of 0 indicates that there exists no correlation at all. For data that can be represented as an AR process, ACF_k will be large for small k and slowly decrease with increasing lag as the correlations between the values decay the farther apart in time the data points are located. On the contrary, due to the assumption that the data points of a MA(1) model are not correlated with any previous values, it can be assumed that the autocorrelations are zero for any $k > 0$. The autocorrelation for $k = 0$ is always 1 as any time-series is perfectly correlated with itself. If however the MA model is of a higher degree, i.e., $q > 1$, then the autocorrelation should be greater than zero for any $k \leq q$ and then "cut off" to zero for any $k > q$. This means that inspecting the ACF allows

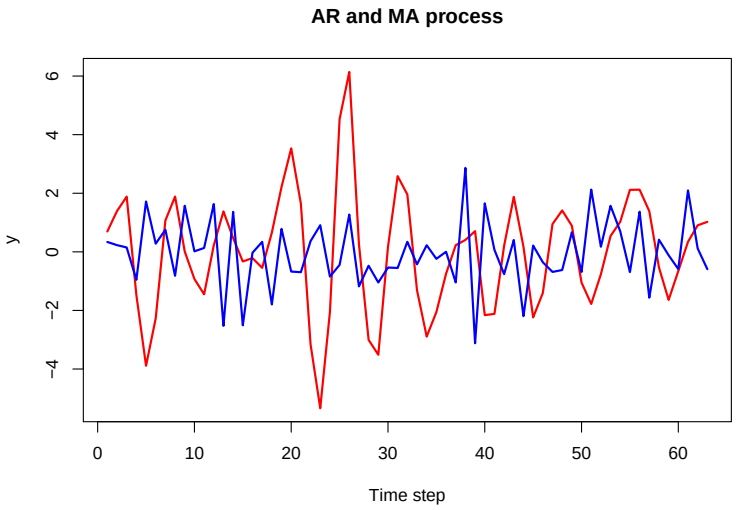

Figure 5.1: Red line: AR(2) process. Blue line: MA(2) process.

determining the degree q of an MA model by observing for which k the autocorrelation drops to zero. In real-world time-series, due to randomness and noise in the data there is typically a confidence value based on the standard error used for determining whether the autocorrelation is small enough to be considered zero or not.

In order to obtain a function that allows the estimation of the degree p of an AR process, it must first be noted that due to the correlations being transferred from y_t to y_{t-1} and from y_{t-1} to y_{t-2} and so on, the resulting correlation between y_t and y_{t-k} is influenced by the data points lying in between. Therefore, the effect of the intervening variables $y_{t-1}, y_{t-2}, ..., y_{t-k+1}$ needs to be removed. This is accomplished by first fitting a linear model $\beta_0+\beta_1 y_{t-1}+\beta_2 y_{t-2}+...+\beta_{k-1} y_{t-k+1}$ that predicts y_t and then fitting a linear model $\gamma_0 + \gamma y_{t-k+1} + \gamma y_{t-k+2} + ... + \gamma k - 1 y_{t-1}$ that predicts y_{t-k}. The partial autocorrelation function (PACF) is then defined as the correlation between the corresponding residuals, i.e.,

$$\begin{aligned} PACF_k = Corr(&y_t - \beta_0 - \beta_1 y_{t-1} - \beta_2 y_{t-2} - ... - \beta_{k-1} y_{t-k+1}, \\ &y_{t-k} - \gamma_0 - \gamma_1 y_{t-k+1} - \gamma_2 y_{t-k+2} - ... - \gamma_{k-1} y_{t-1}) \end{aligned} \tag{5.4}$$

The resulting values will again lie in the interval $[-1, 1]$. The PACF shows opposite behavior regarding AR and MA processes than the ACF. While the partial autocorrelation values slowly decay for any MA process, partial autocorrelations of approximately zero can be observed for all $k > p$ when considering an AR(p) model. Again this behavior can be used to determine the degree p of a model by observing the cutoff point. Note that the partial autocorrelation for lag $k = 0$ is typically set to 1 by convention.

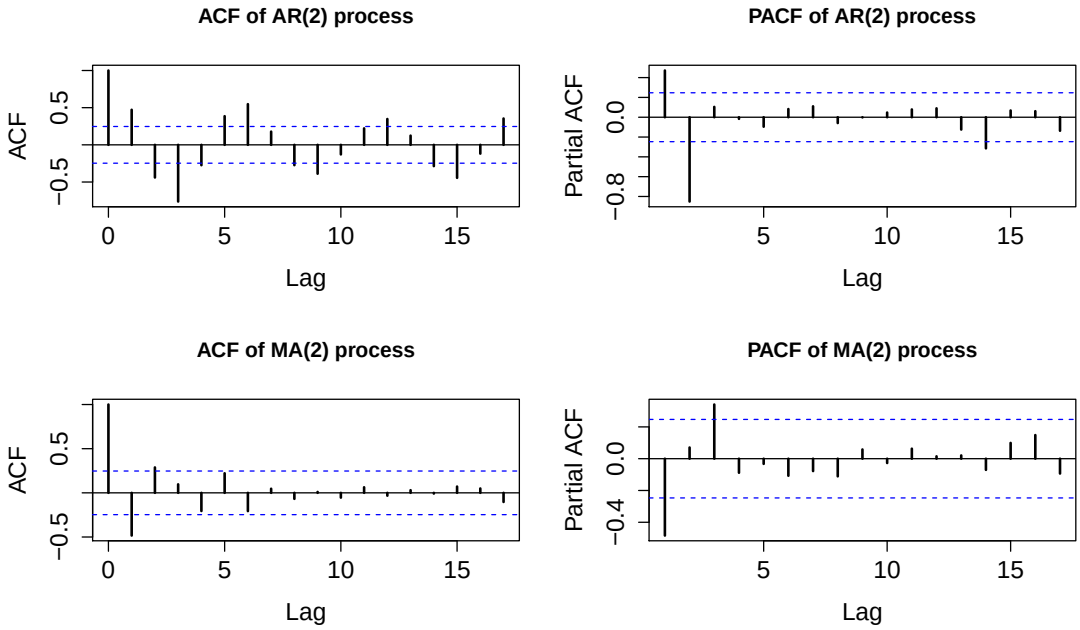

Figure 5.2: ACF and PACF plots of AR(2) and MA(2) processes.

Considering the two sample time-series displayed in Fig. 5.1 that were created based on AR(2) and MA(2) processes, it can be seen that there are no obvious characteristics that make it easy to determine the type of process and its degree that best fits the values and it is therefore necessary to consult ACF and PACF. It is common to plot these measures as a function of k in order to aid the visual reasoning of the selection of a model and its parameters, resulting in the so-called correlograms that are displayed in Fig. 5.2. The previously described characteristics can now be observed in the correlograms, i.e., the ACF of the AR process slowly decays and the PACF shows a cutoff for $k > 2$, the ACF of the MA process also cuts off for $k > 2$ and the PACF slowly decays. As expected, these observations correctly suggest the types of the respective processes as well as their degree.

Neither AR nor MA processes are usually sufficient for an appropriate representation of real-world time series if they are used alone. Combining both processes however often results in a very general model that is able to approximate real data well. This combination is denoted as an ARMA(p,q) model that can be expressed as

$$y_t = a + \phi_1 y_{t-1} + \phi_2 y_{t-2} + ... + \phi_p y_{t-p} + e_t + \mu - \theta_1 e_{t-1} - \theta_2 e_{t-2} - ... - \theta_q e_{t-q} \quad (5.5)$$

ARMA processes are centered around a constant mean and are thus called stationary. However, actual time-series often exhibit some kind of trends or other non-stationary behavior and therefore there is a need to incorporate non-deterministic influences into the model. One way to do this is to find a way to eliminate the non-stationary effects from a

sequence of values so that the resulting time-series can appropriately be approximated with an ARMA process. This can be achieved by differencing, i.e., replacing all y_t with $y_t - y_{t-1}$ for all t, resulting in

$$\begin{aligned} y_t - y_{t-1} = a + \phi_1 (y_{t-1} - y_{t-2}) + \phi_2 (y_{t-2} - y_{t-3}) + ... + \phi_p (y_{t-p} - y_{t-p-1}) \\ + \mu + e_t - \theta_1 e_{t-1} - \theta_2 e_{t-2} - ... - \theta_q e_{t-q} \end{aligned} \tag{5.6}$$

which is equivalent to

$$\begin{aligned} y_t = a + (1 + \phi_1) y_{t-1} + (\phi_2 - \phi_1) y_{t-2} + ... + (\phi_p - \phi_{p-1}) y_{t-p} - \phi_p y_{t-p-1} \\ + \mu + e_t - \theta_1 e_{t-1} - \theta_2 e_{t-2} - ... - \theta_q e_{t-q} \end{aligned} \tag{5.7}$$

Taking the first difference as shown above can be sufficient in many cases, but some time series may require an even higher degree of differencing. For the second order difference, $(y_t - y_{t-1}) - (y_{t-1} - y_{t-2})$ which is equivalent to $y_t - 2y_{t-1} + y_{t-2}$ needs to be computed. Higher order differencing can be realized analogously, although it is rarely necessary in practice. The general model with a variable differencing degree d is denoted as an autoregressive integrated moving-average model of orders p, d and q, or ARIMA(p,d,q) in short.

Besides general trends, many real-world time-series exhibit seasonal effects which are a special kind of non-stationary behavior. Similarly, seasonal influence can be eliminated by differencing, however the periodicity with respect to the number of data points per season needs to be known. With s measured data points per season, the difference $y_t - y_{t-s}$ has to be computed in order to reduce the seasonal effects. Combined with a regular ARIMA process results in a multiplicative seasonal ARIMA process with non-seasonal orders p, d and q and seasonality s with seasonal orders P, D and Q, or SARIMA(p,d,q)$\times$(P,D,Q)$_s$ in short.

5.2 Forecasting

Once an appropriate model that successfully approximates the time-series up to a certain point has been found, the properties of this model can be used to extrapolate over the last recorded time step and thus create a forecast for upcoming values (Cryer and Chan, 2008). This procedure is known as one-step-ahead prediction as it aims at approximating an unknown future value $\hat{y}_{t+1}$ that follows directly after the most recent point y_t. This can be achieved by increasing all variables t by 1, resulting in

$$\hat{y}_{t+1} = a + \phi_1 y_t + \phi_2 y_{t-1} + ... + \phi_p y_{t-p+1} \tag{5.8}$$

for a simple AR(p) process. Note that the error term is omitted from the formula, as it was assumed that all error terms are independent from the previous measurements and the expected value of an unknown error term is 0, i.e., $e_{t+1} = 0$. In order to forecast values that are multiple steps ahead of the last available data point, one-step-ahead

prediction can be applied recursively so that in each step, the value computed in the previous estimation is used as the final data point. Using a variable prediction horizon called the lead time l the AR(p) forecasting model can therefore be expressed as the more generalized model

$$\hat{y}_{t+l} = a + \phi_1 y_{t+l-1} + \phi_2 y_{t+l-2} + ... + \phi_p y_{t+l-p} \tag{5.9}$$

Note that all y_{t+l-j} are actually predicted values $\hat{y}_{t+l-j}$ with $e_{t+l-j} = 0$ for $l > j$, but are known values with any e_{t+l-j} from the available time-series for $l \leq j$.

Analogous reasoning can be applied to forecasting of MA(q) processes. Again t is replaced by $t + 1$ to form an expression for a one-step-ahead prediction model, that is

$$\hat{y}_{t+1} = \mu - \theta_1 e_t - \theta_2 e_{t-1} - ... - \theta_q e_{t-q+1} \tag{5.10}$$

Note that $e_{t+1} = 0$ and was thus omitted from the equation. Just as before, this can be generalized to the following equation supporting a variable amount of lead time l :

$$\hat{y}_{t+l} = \mu - \theta_1 e_{t+l-1} - \theta_2 e_{t+l-2} - ... - \theta_q e_{t+l-p}, \tag{5.11}$$

where again the errors of the predicted values are 0 due to $e_{t+l-j} = 0$ for $l > j$.

Combining the generalized formulations of the AR(p) and MA(q) prediction models for an arbitrary lead time l results in the ARMA(p,q) prediction model

$$\hat{y}_{t+l} = a+\phi_1 y_{t+l-1}+\phi_2 y_{t+l-2}+...+\phi_p y_{t+l-p}+\mu-\theta_1 e_{t+l-1}-\theta_2 e_{t+l-2}-...-\theta_q e_{t+l-p} \tag{5.12}$$

and its extension to the ARIMA(p,d,q) prediction model

$$\begin{aligned}\hat{y}_{t+l} = a &+ (1+\phi_1)\, y_{t+l-1} + (\phi_2 - \phi_1)\, y_{t+l-2} + ... + (\phi_p - \phi_{p-1})\, y_{t+l-p} \\ &- \phi_p y_{t+l-p-1} + \mu - \theta_1 e_{t+l-1} - \theta_2 e_{t+l-2} - ... - \theta_q e_{t+l-q},\end{aligned} \tag{5.13}$$

where just as before $e_{t+l-j} = 0$ for $l > j$.

These computations estimate the future mean value that is expected for any time-series. However, a proper forecast usually requires an additional measure of spread that gives information about the trust in these predictions. When dealing with random variables that follow an unknown distribution, a normal distribution is typically used to assign a variance to the parameter in order to express how much deviation can be expected for a given confidence level. The result is a confidence interval that contains the unobservable true parameter with the specified probability. The same idea can be applied to the forecasts of time-series, however a crucial difference between confidence and prediction intervals must be noted: Prediction intervals are associated with an unknown random variable rather than a parameter of the distribution. The prediction interval that is used in the following therefore contains the actual future value with the specified probability (Hyndman, 2013).

At first, the error of one-step-ahead forecast is defined as

$$e_{t+1} = y_{t+1} - \hat{y}_{t+1} \tag{5.14}$$

This formula is extended for an arbitrary lead time l by

$$e_{t+l} = (y_{t+1} - \hat{y}_{t+1}) + (y_{t+2} - \hat{y}_{t+2}) + ... + (y_{t+l} - \hat{y}_{t+l}) \tag{5.15}$$

and further simplifies to

$$e_{t+l} = e_{t+1} + e_{t+2} + ... + e_{t+l} \tag{5.16}$$

which is proven by displaying the forecasting model as a MA(∞) process (Cryer and Chan, 2008). For a given prediction level α and assuming the errors to be independent and normally distributed, the two-tailed standard normal distribution score $\mathcal{Z}_{1-\frac{\alpha}{2}}$ predicts that the future value is expected to fall within the limits

$$\hat{y}_{t+l} \pm \mathcal{Z}_{1-\frac{\alpha}{2}} \sqrt{Var(e_{t+l})} \tag{5.17}$$

with $(1-\alpha)\,\%$ confidence. Clearly $Var(e_{t+l})$ is unknown as the future values y_{t+l} are not observed yet and thus the actual error values cannot be computed. However, the variance is assumed to be constant over time and can thus be estimated from past values.

Checking whether a future value lies within the prediction interval is an effective method for detecting contextual anomalies, i.e., data points that are anomalous with respect to their local neighborhood. Especially when the one-step ahead prediction interval is computed in every time step, the most recent measured actual value can be compared with the interval from one step before. In short, y_t is identified as an anomaly if

$$y_t \notin \left[\hat{y}_t - \mathcal{Z}_{1-\frac{\alpha}{2}} \sqrt{Var(e)}, \hat{y}_t + \mathcal{Z}_{1-\frac{\alpha}{2}} \sqrt{Var(e)}\right] \tag{5.18}$$

5.2.1 Calculated Example

For a better understanding of the theoretically discussed models and the procedure that leads to the detection of anomalies, a practical example is calculated in a step-by-step manner. For this, a sample time-series shown in Fig. 5.3 is used. The figure shows the measured value y (solid line) as well as the one-step ahead prediction limits that were computed in each step (dashed lines). Furthermore, measured values that exceed the prediction limits were marked by red circles as anomalies. Due to the one-step ahead predictions, it can be seen that y always lags 1 step behind, e.g., the final value y_{25} is used to make a prediction for the following value y_{26} that is unknown at that time.

As an example, the computations of the prediction limits for time step 17 are discussed. This means that the cluster sizes from time steps $1, ..., 16$ are considered for the forecast. At first, an ARIMA model is fitted. Due to the simplicity of this time-series, no components for periodic behavior are required and further $AR = 0, I = 0$ and $MA = 0$ yield the best fit as this setting minimizes the Akaike Information Criterion ($AIC = 130.52$).

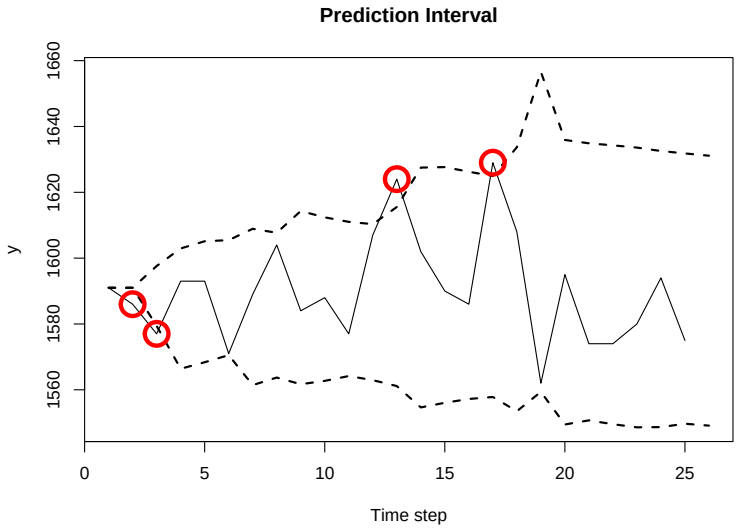

Figure 5.3: A sample time-series. Solid line: Actual measured values. Dashed lines: Computed upper and lower prediction limits. Red circles: Anomalies

This means that there are no autoregressive (AR), integrated (I) or moving average (MA) terms that have to be considered and the prediction is simply the mean, i.e., $\hat{y}_{17} = \frac{1}{16}\sum_{i=1}^{16} y_i = 1591.375$. Furthermore, the variance of the time-series up to this point is $Var(e) = \frac{1}{16-1}\sum_{i=1}^{16}(y_i - 1591.375)^2 = 169.7167$. These informations are used to forecast the limits for a prediction level of 0.99, i.e., $\alpha = 0.01$. This is done by

$$
\begin{aligned}
y_{17} &\in \left[\hat{y}_{17} - \mathcal{Z}_{1-\frac{\alpha}{2}}\sqrt{Var(e)}, \hat{y}_{17} + \mathcal{Z}_{1-\frac{\alpha}{2}}\sqrt{Var(e)}\right] \qquad (5.19)\\
y_{17} &\in \left[1591.375 - \mathcal{Z}_{1-\frac{0.01}{2}}\sqrt{169.7167}, 1591.375 + \mathcal{Z}_{1-\frac{0.01}{2}}\sqrt{169.7167}\right]\\
y_{17} &\in \left[1591.375 - \mathcal{Z}_{0.995}\sqrt{169.7167}, 1591.375 + \mathcal{Z}_{0.995}\sqrt{169.7167}\right]\\
y_{17} &\in [1591.375 - 2.575829304 \cdot 13.02753, 1591.375 + 2.575829304 \cdot 13.02753]\\
y_{17} &\in [1557.818, 1624.932]
\end{aligned}
$$

However, the actual measured value turns out to be $y_{17} = 1629$ and therefore exceeds the upper limit, creating an anomaly. Clearly, longer time-series are needed in order to ensure that proper estimates for the model parameters are determined. Analogous computations are carried out for any point of the time series, resulting in the "tube" that follows the patterns of the cluster size. Fitting the ARIMA model for this example was performed with the R command "auto.arima", which tries to minimize either AIC or BIC by varying the degrees of the AR, I and MA terms. It should be noted that the found values for these terms may change in every time step.

5.3 Correlation

For any given two time-series it can be of interest how they relate to each other and whether they can be considered similar. There are clearly many different characteristics that can be taken into account for that comparison and it usually depends on application-specific purposes. For example, it could be required that the values of the two time-series lie approximately in the same range and that adding or multiplying all values of one series with a constant factor would thus decrease their similarity. On the other hand, it could be of interest whether the same short-term events such as spikes and other prominent features can be found in the place and correct order in both time-series. Furthermore, the periodicity of the signals or the frequency of occurring events in the series can also be indicators for similarity. Finally, long-term trends that are apparent in both time-series can be taken into account as a measure to determine their similarity.

A common measure of similarity that incorporates several of the previously mentioned characteristics is the correlation, which is frequently used in statistics to describe the relationship between two random variables. In general, two variables correlate with each other if their correlation coefficient is larger than 0 and they correlate perfectly if their correlation coefficient is 1, i.e., a change of one variable in any direction indicates a change of the other variable in the same direction. Analogously, for a negative correlation the variables change in opposite directions and a correlation of 0 indicates that there is no relationship between the variables. This principle can also be applied to time-series rather than variables, leading to the cross-correlation function (CCF) (Cryer and Chan, 2008). In the basic case, the CCF can be used to check if the two time series follow a common pattern, i.e., if the slopes from one step to another correspond in each series, if they show an inverted behavior or if there is no relationship at all. In some real-world scenarios, one of the time-series may lag behind the other and thus it needs to be shifted in time in order to produce the correct correlation coefficient for a certain lag. The CCF can be computed for the two time-series $y_1, y_2, ..., y_N$ and $z_1, z_2, ..., z_N$ and any lag k by

$$CCF_k = \begin{cases} \frac{\sum_{t=k+1}^{N}(y_t-\bar{y})\cdot(z_{t-k}-\bar{z})}{\sqrt{\sum_{t=1}^{N}(y_t-\bar{y})^2}\sqrt{\sum_{t=1}^{N}(z_t-\bar{z})^2}} & \text{if } k \geq 0 \\ \frac{\sum_{t=1}^{N+k}(y_t-\bar{y})\cdot(z_{t-k}-\bar{z})}{\sqrt{\sum_{t=1}^{N}(y_t-\bar{y})^2}\sqrt{\sum_{t=1}^{N}(z_t-\bar{z})^2}} & \text{if } k < 0 \end{cases} \tag{5.20}$$

The ACF from Eq. (5.3) is a special case of the CCF where $z_t = y_t$ for all t, i.e., where the correlation between a time-series and itself is computed. Note that it is not necessary to define the ACF for $k < 0$, as a negative shift in time always leads to the same pairs of values being used for the computation as a positive shift. This is due to the fact that the order of the elements is not important for identical time-series, i.e. $(y_t, y_{t-k}) = (y_{t-k}, y_t)$. However, for two different processes the constellation between the compared pairs of values is depending on which time-series lags behind and which one is leading, i.e., $(y_t, z_{t-k}) \neq (y_{t-k}, z_t)$.

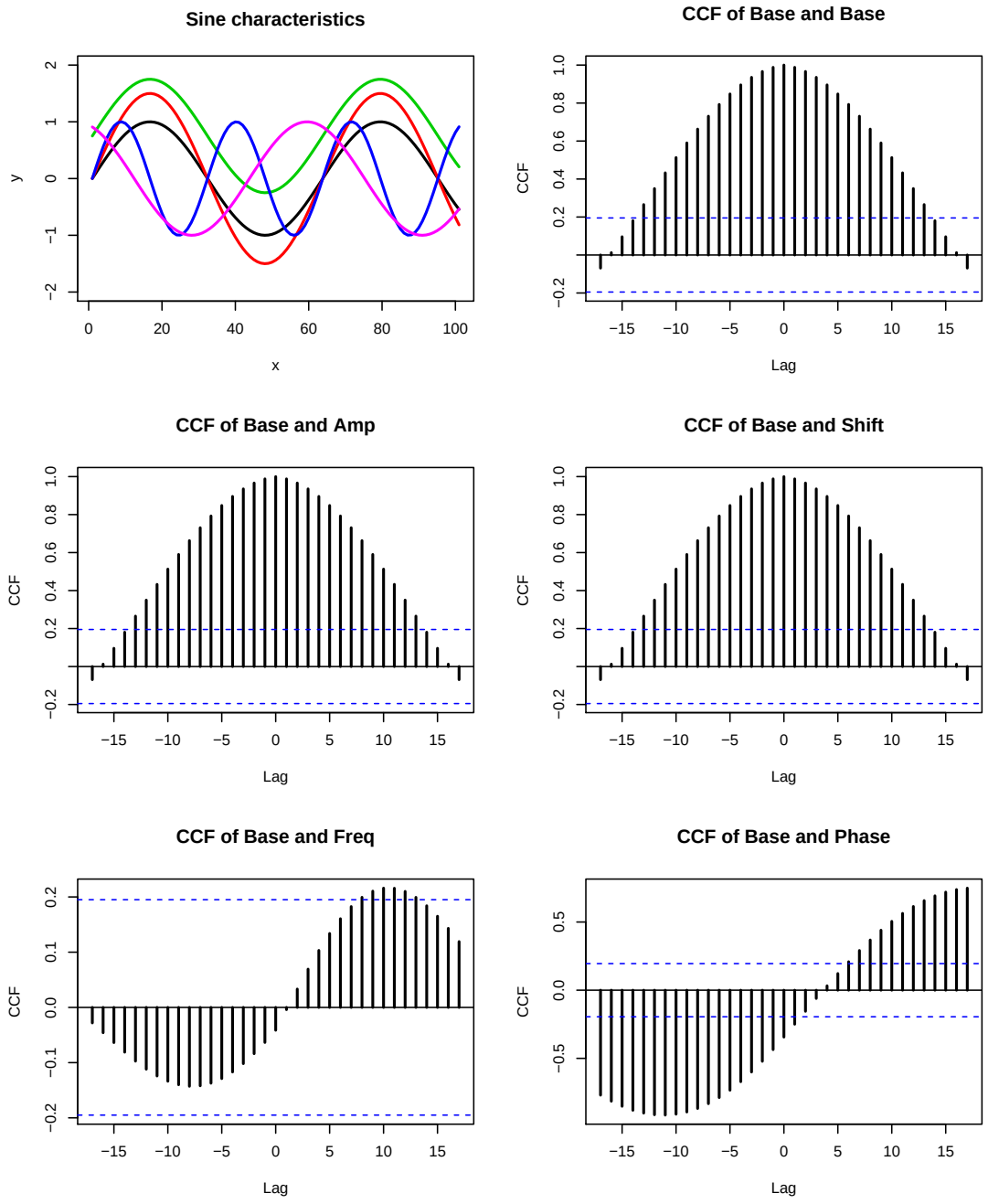

Figure 5.4: Sample sine waves that exhibit different characteristics and CCFs of a base sine wave and its changes regarding amplitude, vertical shift, frequency and horizontal phase-shift.

At the beginning of this section, several properties that could be relevant for determining the correlation are mentioned. In order to visually summarize the characteristics that influence the CCF, the sample sine waves that can be seen in the top-left plot in Fig. 5.4 are considered for an exemplary comparison. The sine waves exhibit the following characteristics: Unchanged base for comparison (black), amplitude changed by multiplication with constant value (red), vertically shifted with constant value (green), increased frequency (blue) and horizontally phase-shifted with constant value (pink). Similar to the ACF and PACF plots, the correlation is plotted for several values of k, where also negative lags are considered. Each sine wave was correlated with the unchanged base and the resulting CCFs for several lags can be seen in the remaining plots of Fig. 5.4. In the top-right plot, the CCF of the base sine wave and itself is plotted which shows that the sine waves achieve a perfect correlation score of 1 for $k = 0$, a characteristic that every time-series that is correlated with itself exhibits. Note that this plot is equivalent to an ACF plot but is mirrored around lag 0. As it can be seen in the following plots, both changes in amplitude and vertical shift do not influence the CCF as there is no difference in the correlation scores compared to the plot where the base sine wave is correlated to itself. A change in frequency that can be seen in the bottom-left plot drastically decreases the correlation coefficient as the slopes of the sine waves do not fit together due to the compressed structure that the blue sine wave exhibits. Finally, the change in phase that can be seen in the bottom-right plot causes that the perfect correlation score of 1 is now reached for a lag different than 0, which can be explained by the fact that the pink sine wave needs to be shifted either back or forth in time in order to match the base sine wave.

5.3.1 Calculated Example

Again, a calculated example is provided for a practical demonstration. For this, two sample time-series Y and Z are used. Figure 5.5 shows their respective progression. It is visible that time-series Z correlates with time-series Y during time steps $1, ..., 10$, even though there is an offset and the slopes do not perfectly match in most of the steps. Computing their respective means yields $\bar{y} = 1587.6$ for time-series Y and $\bar{z} = 1620.5$ for time-series Z. For simplicity, correlation is only considered with lag $k = 0$ in this example. The correlation is computed by

$$\begin{aligned} CCF_0 &= \frac{\sum_{t=1}^{10} (y_t - \bar{y}) \cdot (z_t - \bar{z})}{\sqrt{\sum_{t=1}^{10} (y_t - \bar{y})^2} \sqrt{\sum_{t=1}^{10} (z_t - \bar{z})^2}} \\ &= \frac{(1591 - \bar{y}) \cdot (1630 - \bar{z}) + ... + (1588 - \bar{y}) \cdot (1622 - \bar{z})}{\sqrt{(1591 - \bar{y})^2 + ... + (1588 - \bar{y})^2} \sqrt{(1630 - \bar{z})^2 + ... + (1622 - \bar{z})^2}} \\ &= \frac{3.4 \cdot 9.5 + ... + 0.4 \cdot 1.5}{\sqrt{3.4^2 + ... + 0.4^2} \sqrt{9.5^2 + ... + 1.5^2}} \\ &= \frac{691}{\sqrt{744.4} \sqrt{976.5}} \approx 0.81 \end{aligned} \tag{5.21}$$

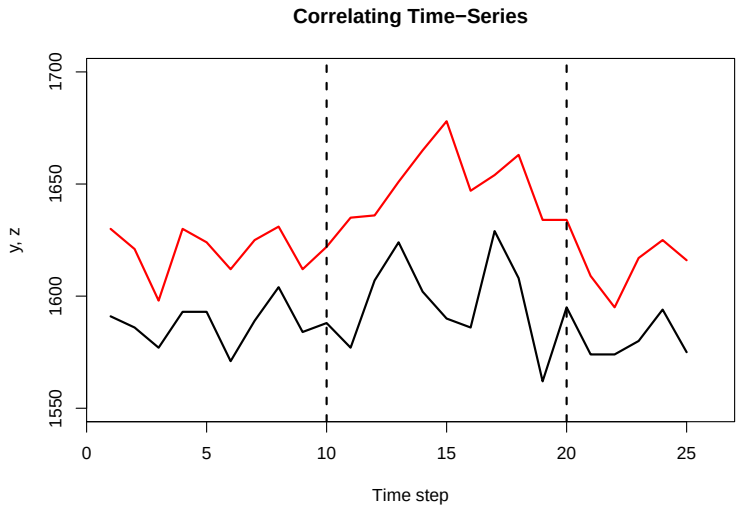

Figure 5.5: Time-series Y (black line) and Z (red line) that correlate between time step $1, ..., 10$ and stop correlating afterwards.

The correlation between time steps $11, ..., 20$ is computed analogously. Note that the respective means of the two time-series have to be computed again as well before computing the correlation. The visualization already suggests that there is less correlation in this interval, and accordingly the computed correlation is only around 0.34. It may be suspicious that time-series that correlate for a long time stop correlating at one point. The reduction of the correlation from 0.81 to 0.34 may therefore be detected as an anomaly.

5.4 Robust Filtering

The issue of an often negative influence of outliers on the quality of forecasts has already been mentioned in Section 2.2. It may therefore be beneficial to reduce the effect of single outlying points on the prediction of future values. Methods that are able to handle noisy data and still produce appropriate results that are mostly independent from the amount and magnitude of the contained outliers are called robust. Examples of robust methods can be found in simple statistical measures that operate on sets of values, for example, the median which orders the input data by size and selects the central value is not influenced by a single strongly deviating value of any magnitude and therefore poses a robust alternative to the mean. The fraction of outliers that can be contained in the data while still producing sensible results is an important characteristic and is usually known as the breakdown point of an estimator. As already mentioned, the non-robust mean has a breakdown point of 0 as adding a single value with a sufficiently large deviation from the actual mean of a given set of values is able to arbitrarily alter the mean of the

combined values. In contrast to that, the median is able to achieve the highest possible breakdown point of 0.5 which means that a reasonable result will be computed even if half of the involved values consists of outliers.

These issues also play a role in the prediction of time-series models. When considering a simple ARIMA model, it is clear that the appearance of an outlier has a drastic influence on the forecast of the following value as the prediction will follow the direction of the outlier and possibly overshoot the actual value occurring in the following time step. On the other hand, the large error occuring in the time step of the outlier and possibly succeeding steps will cause the variance to increase. It then becomes more likely that the following actual values still lie within the prediction ranges even though they might be anomalies that are rather far located from the previous data points. In any way it is sufficient for practical purposes if the very first appearing outlier is reported as an anomaly independent from whether all future values correctly follow the previous trend as it would have been expected if the outlier had never occurred. Due to this necessity of performing a manual inspection of the involved clusters in the case of a detected anomaly, increasing the robustness of the ARIMA models is not of primary focus.

It can however be beneficial to additionally use a robust filter that operates on the time-series as a whole and does not rely on one-step ahead forecasts. This filter is able to identify outliers independent from the ARIMA prediction models and thereby detects anomalies that lie within the prediction boundaries and are unnoticed by the ARIMA models. Moreover, the robust method is able to confirm that the data points that lie outside of the prediction limits by also detecting them as anomalous. Such a method that is originally designed for monitoring medical data was introduced by Fried (2004). In this work, a time window of fixed size $\omega = 2w + 1$, for any predefined $w > 0$, is created for any point y_t of the time-series which is approximated by a linear model with level μ_t and slope β_t via

$$\hat{y}_{t+i} = \mu_t + i\beta_t + E_{t+i} + e_{t+i}, \quad i = -w, -w+1, ..., w, \tag{5.22}$$

where E_{t+i} is a random noise and e_{t+i} is the error of the approximation. If ω is supposed to be even, the formula needs to be adapted accordingly as the time window would not be symmetric around y_t anymore and further the start and end of the time-series where $t - i < 1$ and $t + i > N$ need to be especially considered. The author suggests two different estimators for computing μ_t and β_t: The least median of squares estimator

$$T_{LMS} = argmin \left\{ (\mu, \beta) : median \left(y_{t+i} - \mu - i\beta \right)^2 \right\} \tag{5.23}$$

and the repeated median estimator $T_{RM} = (\hat{\mu}_t, \hat{\beta}_t)$ with

$$\hat{\mu}_t = med_i \left(med_{j \neq i} \frac{y_{t+i} - y_{t+j}}{i - j} \right) \tag{5.24}$$

$$\hat{\beta}_t = med_i \left(y_{t+i} - i\hat{\beta}_t \right) \tag{5.25}$$

Both of them have a breakdown point of $\lfloor \omega/2 \rfloor / \omega = 0.5$ which is optimal for time-series. There also exist several methods for approximating the variance σ_t in a robust way, one of the most simple is taking the median of the residuals $r_i = y_{t+i} - \hat{y}_t - \hat{\beta}_t i$, i.e.,

$$\hat{\sigma_t} = c \cdot med \{|r_{t-w}|, ..., |r_{t+w}|\}, \tag{5.26}$$

where c is a constant that depends on the size of the time window. Similar to the outlier detection in the ARIMA model, a simple extrapolation of each time window allows computing an estimation of the future value which is in turn compared with the actual value. The main difference is that this was done incrementally in the ARIMA model, i.e, the prediction was stored until the cluster evolution process was carried out for the next time step and only then the comparison with the actual value was possible. As the robust filter method operates on a finite and known sequence of values the error is computed between the incoming value and its estimation in the same step. Similar to the comparison done in the ARIMA model, a prediction range based on the estimation of the variance is computed. In detail, an outlier is found if $|r_{i+1}| > c_0 \hat{\sigma}_t$, where c_0 is a predefined factor.

The procedure of the algorithm involves iteratively estimating the parameters μ_t, β_t and the error variance σ_t in order to detect outliers in every time window centered around t. The variable t is thereby increased until the end of the time-series is reached. The robustness of this method comes from replacing the outliers with an estimate $\hat{y}_{t+i+1} = \hat{\mu}_t + (i+1)\hat{\sigma}_t + c_1 sgn(r_{i+1})\hat{\sigma}_t$ where $sgn(.)$ is the signum function and c_1 is a factor determining the influence an outlier has on the replacement.

Another feature of the algorithm proposed in the paper is the detection of level shifts that is considered separately and also contributes to the robustness of the method. This is done by determining the fraction of values on the right side of each time window $y_t, y_{t+1}, ..., y_{t+w}$ that falls outside of a prediction boundary again defined by a constant d_2 in combination with the estimation of the variance. If this is the case for more than half of the residuals, a level shift has been detected. In mathematical terms, the level shift is detected if

$$\sum_{j=-w}^{w} I_{\{r_j > d_2 \hat{\sigma}\}} > \sum_{j=-w}^{w} I_{\{r_j \leq d_2 \hat{\sigma}\}} \tag{5.27}$$

As only the second half of the time window is considered, the breakdown point is $(w/2)/\omega \approx 0.25$. The level shift detection can also be implemented in the previously mentioned algorithm that slides the time window over the time-series. For that, whenever a level shift is detected, the trend before the shift is extrapolated right before the shift and then the algorithm is restarted w steps after the shift so that more than half of the values lie already on the new level.

5.5 Multivariate Outlier Detection

Several cluster features and evolution metrics have been mentioned in Sections 4.2 and 4.3. As each metric is only able to display information about a certain characteristic

and it may not be possible to come to the same conclusions about cluster developments judging from any other metric, it is desirable to involve as many of the metrics as possible when detecting anomalies. All of them are suitable to be represented as a time-series and can thus be used for fitting ARIMA models or performing a correlation analysis as it was explained in the previous sections.

It is not clear whether an anomaly should be reported if it is occurring just in a single metric but not in any of the others. This is due to the fact that the previously mentioned techniques report anomalies based on predefined thresholds or probabilities, but they do not offer a suitable way of combining the results of time-series from different metrics. Furthermore it cannot be assumed that each metric is of equal importance when detecting outliers and an appropriate weighting scheme would be required.

Therefore, instead of individually considering each of these additional time-series that can be associated with single clusters, a multivariate approach is able to identify outliers in a higher dimensional space by combining the influence of all features at the same time. This combination reduces the amount of required thresholds that are necessary to determine whether a point is an outlier or not due to the fact that each feature contributes to the position of the point in space simultaneously. This means that while it is possible to detect an anomaly if one of its features is highly anomalous, also the combined effect of only smaller deviations that are present in multiple features is sufficient for identifying this point as an outlier.

An efficient algorithm for outlier detection in high dimensions is introduced by Filzmoser et al. (2008). The proposed approach emphasizes robustness by rescaling the N samples using the median and MAD (median absolute deviation) and tackles the problem of exponentially increasing computation time that is prevalent in other existing outlier detection algorithms when operating in higher dimensions by applying a principal components analysis (PCA). This not only effectively reduces the amount of dimensions from p to p^* while ensuring that the largest possible amount of information is contained in the principal components, outliers are also likely to stick out even more due to the fact that they increase the variance along their coordinate and are thus more likely to partially determine the direction of the principal components. Another advantage of this method is that the Euclidean norm computed in the principal component space is equivalent to the Mahalanobis distance in the original space, but is easier to compute. Once the most relevant principal components of the PCA-transformed data z_{ij} have been selected, a robust kurtosis measure computed by

$$w_j = \left| \frac{1}{N} \sum_{i=1}^{N} \frac{(z_{ij} - med_i z_{ij})^4}{MAD\,(z_{ij} - med_i z_{ij})^4} - 3 \right| \tag{5.28}$$

is used to weigh each component by the likelihood of revealing outliers. Moreover, a measure of outlyingness is assigned to each sample by applying the translated biweight function on the robust distances d_i that are transformed to fit a $\chi^2_{p^*}$ distribution. Using

$$c = median\,\{d_1, ..., d_N\} + 2.5 \cdot MAD\,\{d_1, ..., d_N\} \tag{5.29}$$

and defining M as the 0.33 quantile of $\{d_1, ..., d_N\}$ the outlyingness is then computed by

$$w_{1i} = \begin{cases} 0 & \text{if } d_i \geq c \\ \left(1 - \left(\frac{d_i - M}{c - M}\right)^2\right)^2 & \text{if } M < d_i < c \\ 1 & \text{if } d_i \leq M \end{cases} \tag{5.30}$$

where values closer to 0 indicate strong outliers regarding their location and values close to 1 indicate normal data points. Furthermore, scatter outliers that possess a different scatter matrix than the rest of the data are effectively detected in the previously described principal component space when the kurtosis weighting scheme is omitted. For this, the robust distances are again transformed to a $\chi^2_{p^*}$ distribution and the weights w_{2i} for each sample are computed by the translated biweight function as before, however setting M^2 to the 0.25 quantile and c^2 to the 0.99 quantile of a $\chi^2_{p^*}$ distribution. Finally, the weights computed for location outliers and the weights computed for the scatter outliers are combined into a final weight by

$$w_i = \frac{(w_{1i} + s) \cdot (w_{2i} + s)}{(1 + s)^2} \tag{5.31}$$

where the scaling constant $s = 0.25$ ensures that $w_i \neq 0$ if only one of the two weights is 0. Again, samples with a weight close to 0 indicate a high outlyingness and a threshold is used to differentiate between outliers and normal data points.

5.6 Algorithm

In order to automate the anomaly detection procedure, an algorithm that involves a specific sequence of steps was developed. This algorithm is based on the incremental clustering approach that was already described in Section 3.4 and applies the cluster evolution techniques described in Chapter 4 on the identified clusters. Then, the procedure further incorporates the time-series analytics previously described in this chapter in order to detect anomalies.

A detailed overview about the steps of the algorithm is given by the flowchart in Fig. 5.6. As it can be seen, steps (1)-(4) describe the previously mentioned incremental clustering algorithm. In step (1), the log lines are either read from a log file or received line by line as a stream. They are initially preprocessed in step (2) in order to ensure that they do not contain any special characters that cannot be represented or compared properly, i.e., all characters outside of the range $[32, 126]$ from the ASCII table are removed. Furthermore, multiple consecutive spaces are reduced to a single space as they do not convey any relevant information and would only increase the distance of otherwise identical lines. Time-series analysis obviously requires the time stamps of the log lines to be extracted in order to check that the processed line still lies within the current time window.

Step (3) describes the construction phase that takes place within each time window. The algorithm iteratively processes the log lines and adds them to the current cluster map

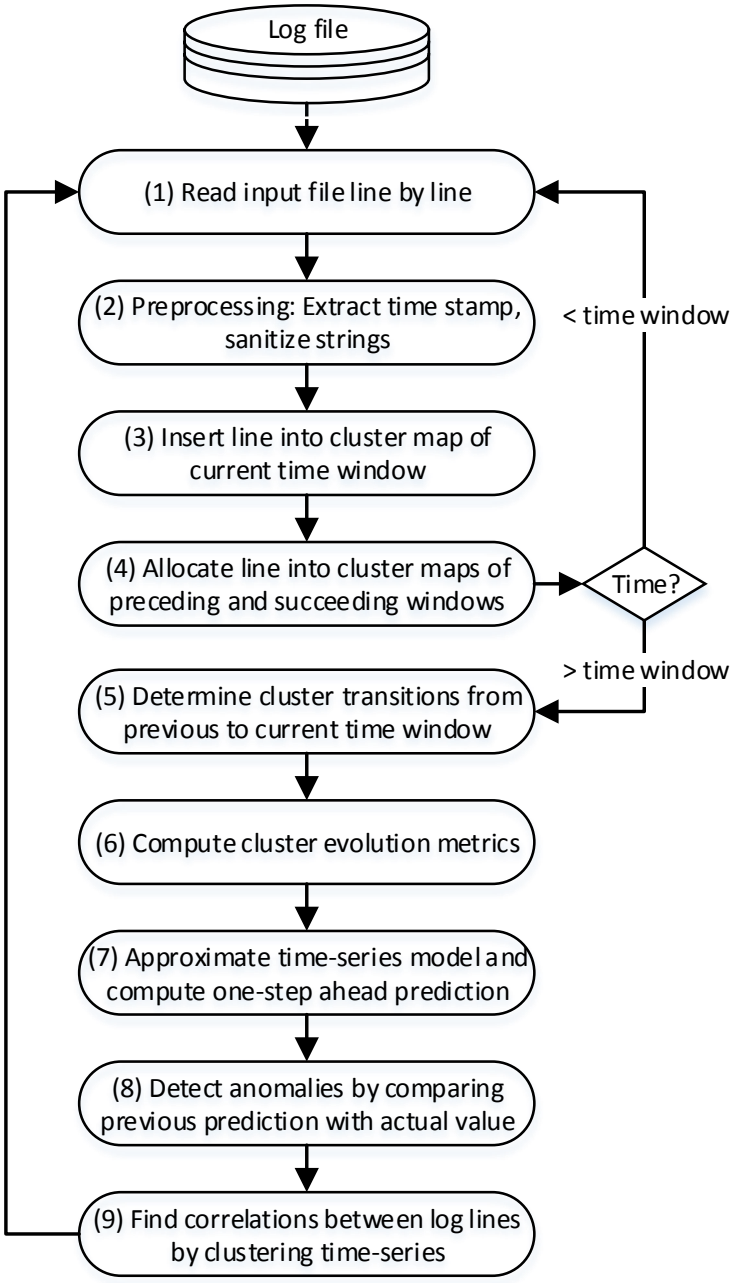

Figure 5.6: Flowchart of the anomaly detection procedure. Steps (1)-(4) involve clustering, steps (5)-(6) involve cluster evolution and steps (7)-(9) involve time-series analysis.

following the incremental clustering approach that uses filters in order to efficiently detect the most similar cluster for any incoming log line. In the following, the cluster map that is established during a specific time window is seen as a static view of the data at that point and is thus referred to as the cluster map of a specific time step rather than a period of time. The ordered sequence of cluster maps can also be seen as static snapshots of the system at specific points in time. However, in order to extract cluster-specific informations from this sequence rather than only being able to consider the cluster maps as a whole without any possibility to reliably find any nested connections between the maps, cluster evolution techniques must be applied. This is accomplished by employing the clustering model introduced in Section 4. According to the model, the allocation phase in step (4) clusters all log lines into the already existing cluster maps from the preceding and succeeding time windows.

Once a time window is completed, i.e., all log lines that occurred within that period are allocated to a cluster in the cluster map, the log lines of that window have also been allocated in the previous map and further the log lines of the previous time window have also been allocated into the current map, step (5) computes the cluster transitions. This is done by computing the overlap coefficient from Eq. (4.2) for any two clusters from neighboring time steps. A high overlap indicates that the cluster from the former time window transformed into the cluster from the later time window, i.e., both of the clusters were generated by the same underlying process and should therefore be connected. More sophisticated changes in cluster development, such as splits or merges, are detected using Algorithm 4.1. After this step is completed, all the connections and relationships between the clusters from the currently finished time step and its preceding time step are known.

In real-world log files time stamps do not necessarily have to be in the correct order, i.e., appearing log lines may have time stamps that lie in the past. This can be caused by components operating with an incorrect system time, wrongly adjusted time zones, system errors, busy processes that take up all computational power or delays caused by slow connections or distributed components. In the most cases, these delays lie within a few seconds and are thus likely to have minor influence as they will still be contained in the same time window, unless they occur precisely after a time window was finished. For other cases where delays exceed a reasonable amount of time it can be difficult to adjust the cluster maps that have already been completed in the past as this would influence the overlaps that were computed. In the worst case, all the connections between clusters from maps that occur later than the lagged time stamp would have to be recomputed as it cannot be guaranteed that the same cluster evolutions would be identified. Due to the fact that such artifacts only occur very rarely and they are likely to have minor influence on the overall cluster evolution analysis as they only make up a small fraction of the totality of log lines, it is suggested to simply overwrite the faulty time stamps with the last correct time stamp that was read previously.

Given the set of cluster connections, step (6) uses this information for computing the evolution metrics, including rates about growth, stability and other cluster features mentioned in Section 4.3. After completing the cluster evolution techniques in steps

(5)-(6), the computed metrics and especially the development of the cluster sizes are then analyzed by time-series analysis methods in steps (7)-(9). In step (7), the cluster sizes are approximated by an ARIMA time-series model. There are different approaches on how to determine the correct parameters of the model and efficiency may play an essential role as each tracked cluster requires a fitted model that is recalculated in every time step. According to the theory explained in this chapter, the models are then used to predict the size of each cluster in the next time step. This is called a one-step ahead prediction as it is performed in every time step and the forecasting horizon is always set to 1. Following this logic, there exist a prediction and a confidence interval for every time step except for the very first one where no preceding values could be processed. Anomalies are detected in step (8) by checking whether the actual values from the currently processed time window lies within the expected ranges estimated from the historic values of the preceding time windows. In the case that the values lie outside of the predicted bounds, an alarm is raised that gives information about the cluster and the time step where the anomaly occurred. Due to the fact that an anomaly can appear at any point during a time window, the average time until it is detected is half of the time window size.

Finally, step (9) describes the correlation analysis. The occurrences and frequencies of some of the log line types correlate over time which means that the time-series of the cluster size developments also correlate with each other. The correlation analysis thus involves clustering together the time-series that share a high correlation coefficient. In order to accomplish this in an efficient way, the clustering is performed in a similar manner as the clustering of the log lines. For that, an initially empty list of sets that contains groups of correlating clusters is created and gradually filled. The first time-series being added to this list obviously forms a new group as there is no other time-series in the list that it could correlate to. Every time-series that forms a new group immediately becomes its representative that is used for the comparison with following time-series. For any further time-series, the correlation between itself and all the representatives existing at that point is computed and the time-series is added to the most similar group if the correlation coefficient exceeds a predefined threshold. Eventually all the time-series are contained in the list either as a representative or as a member of a group.

The correlation analysis in that form already gives interesting details about the relationships between clusters that were previously unnoticed. However, in order to detect anomalies, at least two correlation analyses in different time steps need to be performed as only the change in correlation is considered anomalous. For example, two time-series that consistently correlated with each other over several time steps and spontaneously stop correlating with each other or two time-series without any historic relationships suddenly start correlating with each other indicate a change of the system behavior. Note that it is possible to perform the correlation analysis and compare its results with previous outcomes in every time step just as it was done with the ARIMA predictions. However, due to performance reasons and as the time-series only change in the most recent value that was added, it is generally reasonable that the correlation analysis is only carried out in larger time intervals. The reason for this is that all the points in

the time-series are of equal weight when computing the correlation, while for the value predictions the most recent value had a higher influence on the value lying one-step ahead. Therefore, as all points stay the same except for the most recent one being added and the oldest one being removed, the correlation can be expected to change only insignificantly over short periods.

5.7 Aggregated Detection

The previously explained algorithm specifically aims at detecting anomalies for a specific cluster, i.e., each detected anomaly is associated with exactly one cluster. This clearly has some advantages, e.g., the cluster representative or the cluster members immediately give information about the exact type of log line that is affected by the anomaly and it may therefore be easier to trace back the source of the problem.

However, the enormous amounts of clusters in combination with the probability-based approach of the prediction limits naturally causes a rather high number of false alarms that are raised in each time step. For example, if a prediction interval that contains the actual value with 99% probability is computed for 100 clusters in each time step, 1 false alarm is raised per time step on average. In practical applications it is therefore a tedious task to react to every single alarm that is raised and there is a need for a more robust measure.

An intuitive way to solve this problem is to aggregate the anomalies that occur in each time step. On average, randomly occurring anomalies caused by natural fluctuations and noise should occur uniformly distributed over time and are unlikely to collectively occur in multiple clusters at a single point in time. Given that an actual anomaly usually affects more than 1 cluster, counting the number of clusters that report anomalies is therefore a reasonable start. However, this does not incorporate that an anomaly that lies far outside of the prediction interval should be considered as more anomalous than an anomaly that just barely exceeds the upper or lower limit. An aggregated measure should therefore consider the following cases:

(a) No anomalies detected: In the case that no anomalies are detected in any of the clusters, the aggregated anomaly score should be 0.

(b) Few clusters, small magnitudes: All anomalies occurring in a time step are only reported from one or few clusters and further none of the anomalies lie far outside of the prediction interval. Therefore, the anomaly score of this time step should be low.

(c) Few clusters, large magnitudes: Only one or few clusters report an anomaly, however one or more of these anomalies lie far outside of the prediction interval. The anomaly score of this time step should be moderately high.

(d) Many clusters, small magnitudes: Anomalies are reported from many clusters, however none of the reported anomalies lie far outside of the prediction interval. Again, the anomaly score of this time step should be moderately high.

(e) Many clusters, large magnitudes: Anomalies are reported from many clusters and one or more of them lie far outside of the prediction interval. Clearly, this is the most severe case and thus the corresponding time step should receive a high anomaly score.

Furthermore, not all clusters should be equally weighted when considering the anomalies that are detected in their developments. A cluster that has only recently emerged is likely to report more false alarms due to the fact that too few historic values are available to properly compute the prediction interval. On the other hand, clusters that have been existing for a large number of time steps are more likely to exhibit stabilized features and are therefore more trustworthy. Contributions to the anomaly score of a time step should hence be weighted according to the respective durations that a cluster has already been existing.

Before introducing such an aggregated anomaly score, a value s_t that mirrors anomalous y_t values that fall below the lower limit of the prediction interval to the upper side is defined for convenience. With the upper prediction limit $u_t = \hat{y}_t + \mathcal{Z}_{1-\frac{\alpha}{2}}\sqrt{Var(e)}$ and the lower prediction limit $l_t = \hat{y}_t - \mathcal{Z}_{1-\frac{\alpha}{2}}\sqrt{Var(e)}$, the mirrored value is defined as

$$s_t = \begin{cases} y_t & \text{if } y_t > u_t \\ 2\hat{y}_t - y_t & \text{if } y_t < l_t \end{cases} \tag{5.32}$$

Note that the first case corresponds to y_t lying above the prediction limit, meaning that no action is necessary. The second case corresponds to y_t lying below the prediction limit causing that the point needs to be mirrored around the predicted value $\hat{y}_t$ which is always positioned in the center of the prediction interval. Therefore, the distance between y_t and the closest prediction limit will remain the same after mirroring. Furthermore, the set of clusters containing an anomaly at time step t is defined as

$$\mathcal{C}_A^t = \left\{ C \in \mathcal{C}^t : y_t > u_t \vee y_t < l_t \right\} \tag{5.33}$$

With these definitions and the duration τ_t that measures how many time steps a cluster has already been existing, the anomaly score a_t at time step t is defined as

$$a_t = \begin{cases} 0 & \text{if } \mathcal{C}_A^t = \emptyset \\ 1 - \frac{\sum_{C \in \mathcal{C}_A^t} (u_t \cdot log(\tau_t))}{|\mathcal{C}_A^t| \cdot \sum_{C \in \mathcal{C}_A^t} (s_t \cdot log(\tau_t))} & \text{otherwise} \end{cases} \tag{5.34}$$

Both u_t and s_t are multiplied with the same $log(\tau_t)$ in order to give clusters that have been existing for a longer time more weight. The logarithm was used to dampen this effect.

Note that u_t in the numerator defines the upper limit of the prediction interval and the variable s_t in the denominator represents the actual value. It is known that $s_t > u_t$ due to the fact that only clusters that contain an anomaly at time step t are considered in the sum and actual values $y_t < l_t$ have been mirrored to the upper side. As both terms are weighted equally, the denominator must always be larger than the numerator and therefore the division is guaranteed to be smaller than 1. Larger deviations from the expected value, i.e., a higher value for s_t, hence cause that the division yields values closer to 0.

Furthermore, including the term $|\mathcal{C}_A^t|$ in the denominator accounts for the impact of more clusters reporting anomalies. Again, a higher amount of clusters reporting anomalies draws the resulting value closer to 0. Finally, the result is subtracted from 1 in order to have anomaly scores close to 0 indicating normal behavior while anomaly scores close to 1 indicate anomalous behavior. In practice, an alarm should be raised if the anomaly score exceeds some predefined threshold.

The characteristics of this equation and the computation of the anomaly score are demonstrated in an example. Figure 5.7 shows the developments of 3 measured features from clusters A, B and C as well as the computed prediction intervals and anomalies detected in each time-series. For this simple demonstration it is assumed that only 3 different types of log lines appear in the log file and thus these 3 cluster developments appropriately describe the complete system behavior. The first anomaly occurring in the second steps of every cluster development is ignored in the following as it is caused by the already mentioned issues that appear when too few historic values are used in the ARIMA model.

The remaining anomalies can be related to the previously mentioned cases regarding magnitude and number of affected clusters. Only cluster A is affected by the anomaly occurring at time step 6, which is corresponding to case (b). This is also true for the anomaly detected in cluster C at time step 13, however here the magnitude of the deviation is larger, thus corresponding to case (c). Finally, every cluster reports an anomaly with rather low deviation in time step 19, which is corresponding to case (d). No anomalies are detected in all the remaining time steps, thus corresponding to case (a). Case (e) was omitted for this example. Furthermore, cluster C only emerges at time step 6, meaning that its contribution to the anomaly score is weighted lower compared to clusters A and B in every following time step.

At first, time step 6 is considered. The forecasted upper prediction limit of cluster A computed in time step 5 is 5.81, however the actual value measured in the following time step is 6. The actual value exceeding the upper limit raises an anomaly that is marked in the figure with a red circle. At his point in time, the cluster has been existing since 6 time steps and the anomaly score is therefore computed as

$$a_6 = 1 - \frac{5.81 \cdot log(6)}{1 \cdot 6 \cdot log(6)} = 1 - 0.97 = 0.03 \tag{5.35}$$

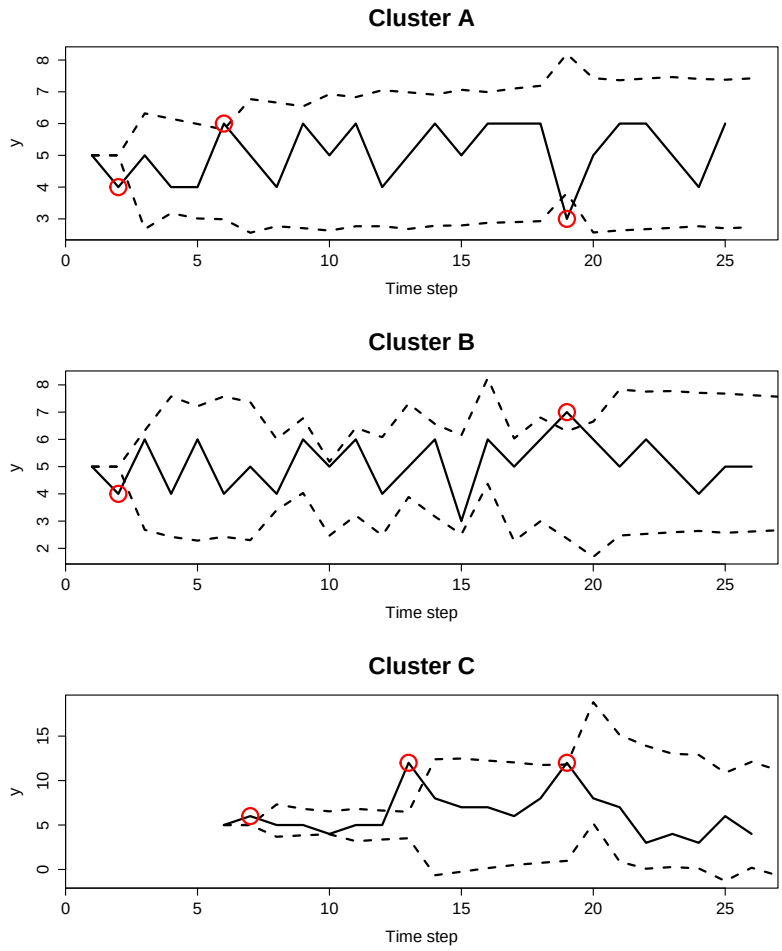

Figure 5.7: Developments of clusters A, B and C, including prediction limits and detected anomalies.

As expected, this is a relatively low score. It is also noteworthy that the weighting term always cancels out in the case that only one cluster is considered. This is also the case when all clusters contained in ${\mathcal{C}_{\mathcal{A}}}^t$ have been existing for the same amount of time. Only if at least two clusters that report anomalies at time step t have been existing for different amounts of time steps, the weighting affects their influence on the anomaly score accordingly.

Next, the anomaly score of time step 13 is computed. The predicted upper limit of cluster C at this time step is 6.49 and the actual value is 12. Furthermore, the cluster came into existence 8 time steps ago. The anomaly score is thus

$$a_{13} = 1 - \frac{6.49 \cdot log(8)}{1 \cdot 12 \cdot log(8)} = 1 - 0.54 = 0.46 \tag{5.36}$$

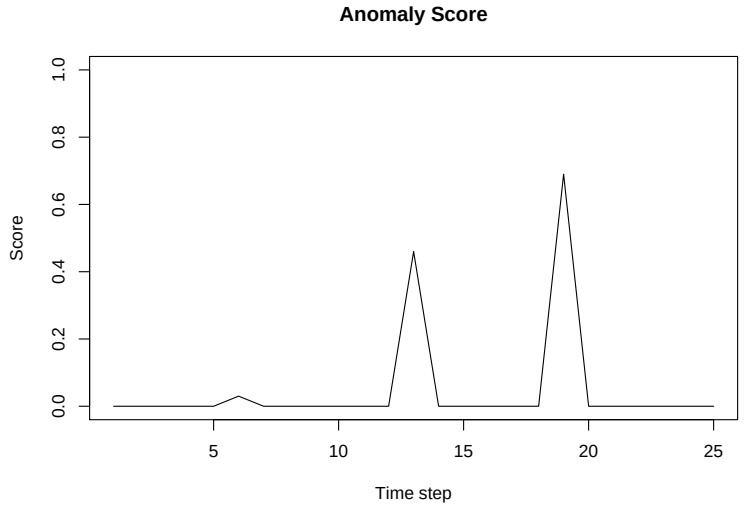

Figure 5.8: Aggregated anomaly score of clusters A, B and C.

This is already a larger score due to the higher deviation from the expected value. The computation of the anomaly score of time step 19 is more complicated as more clusters are involved. Furthermore, the actual value of cluster A is 3 and therefore falls below the lower prediction limit of 3.81, i.e., the second case of Eq. (5.32) holds true. Before computing the anomaly score, the point first needs to be mirrored on the upper side according to the stated equation. With the predicted value $\hat{y}_t = 6$, the new point is $s_t = 2 \cdot 6 - 3 = 9$. Note that the upper prediction limit is 8.19 and thus the distance to the closest prediction limit remains the same, i.e., $|3 - 3.81| = |9 - 8.19| = 0.81$. The upper prediction limits of clusters B and C are 6.30 and 11.80 respectively as opposed to their actual values, 7 and 12. Furthermore, while clusters A and B have been existing since 19 time steps, cluster C has only been existing since 14 time steps. The weighting term $log(\tau_t)$ ensures that the influence of cluster C on the resulting anomaly score is lower compared to the influences of clusters A and B. This leads to the anomaly score

$$\begin{aligned} a_{19} &= 1 - \frac{8.19 \cdot log(19) + 6.30 \cdot log(19) + 11.80 \cdot log(14)}{3 \cdot (9 \cdot log(19) + 7 \cdot log(19) + 12 \cdot log(14))} \\ &= 1 - \frac{73.81}{3 \cdot 78.78} = 1 - 0.31 = 0.69 \end{aligned} \tag{5.37}$$

Due to the fact that the anomaly was recorded in three different clusters, the score is higher than the score computed at time step 13. Finally, the anomaly scores of all time steps where no anomalies are reported are set to 0. Figure 5.8 shows a visualization of the computed anomaly scores that clearly shows the 3 time steps where anomalies occur. As expected, the anomaly scores of time steps 13 and 19 are significantly larger than the anomaly score of time step 6.

CHAPTER 6

Evaluation

As outlined in the introduction of this thesis, unsupervised methods are able to detect anomalies on unlabeled data. As labeled data is rare, this is a beneficial setting when applying such systems in practice. However, a proper evaluation should not solely consist of a subjective and qualitative interpretation of results achieved on a largely unknown data set. This is due to the fact that there is no way to tell whether detected anomalies actually correspond to real anomalies that occurred in the system and whether most of the anomalies in the data have actually been detected. While it would be easy to perform the evaluation on synthetic log data as it was done in Section 4.4.2, one could criticize that this kind of data does not resemble log data as it occurs in the real world and is therefore not appropriate for a realistic evaluation. As a compromise, the evaluation is carried out on a semi-synthetically created log file that only contains a specific amount of anomalies that occur at known points in time. This combines the advantages of the real world data by incorporating sufficient complexity and the advantages of synthetic data by enabling the creation of a ground truth table, i.e., a complete set of anomalous log line types that are known to exhibit certain characteristics at specific time steps.

6.1 Log Data

The generation of the log data was carried out by adapting the approach introduced by Skopik et al. (2014). The setting consists of a MANTIS Bug Tracker System[1] deployed on an Apache web server. A screenshot of the MANTIS user interface can be seen in Fig. 6.1. A variable amount of virtual users simulate real user behavior by navigating on the website. The users perform actions just as real users would do, including reporting, assigning and deleting issues as well as clicking on entries from the task menu and regularly logging in and out. Although there is a script with predefined paths underlying the

[1] MANTIS Bug Tracker available at `https://www.mantisbt.org/`, accessed 08-November-2017

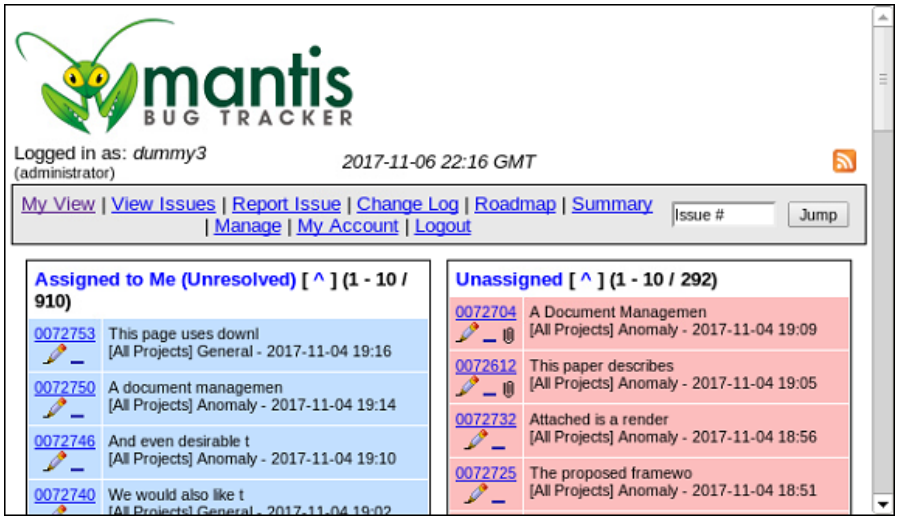

Figure 6.1: Main Page of MANTIS Bug Tracker.

actions of the users, their behavior is highly complex due to random numbers determining which paths are followed and what kind of selections are made in each step. Clicking on a certain button therefore does not always generate an identical set of log lines, especially because they frequently contain the current date or time, IDs as well as random selections, numbers and strings. Therefore, clustering requires the fuzzy matching approach that was explained in Chapter 3. Furthermore, some types of log lines (e.g., “Init DB” and “Quit”) are produced for every single SQL query, while others only occur when a special action is performed. This is an important characteristic as it implies that the caused relative changes are of different magnitude in each cluster. In any way, an action always leads to the generation of a set of log lines and therefore anomalies manifest themselves in multiple clusters.

Logs are recorded from three components: The Web Server, the SQL database and the reverse proxy. The logs therefore contain the accessed URLs, user-specific data such as the MAC address as well as the executed SQL queries. Examples for such lines can be seen in the sample log lines that have already been shown in Fig. 3.3.

With this setup, an illustrative attack scenario is introduced. The scenario takes place over the course of 96 hours (4 days) and was simulated in real-time. Five virtual users are involved in the creation of the logs. Three of them continuously produce normal behavior, i.e., the likelihood of following certain paths in their script remains unchanged. For any given time window that is large enough to cover a reasonable amount of actions, it can therefore be assumed that these users produce a steady average size for all the clusters. One of the remaining users simulates an automatized software or program that operates only in the first 30 minutes in every hour. The actions carried out by this user are also performed by the other users and result in a periodic behavior in the affected clusters. The behavior of these users is considered to be anomaly-free, i.e., in the optimal

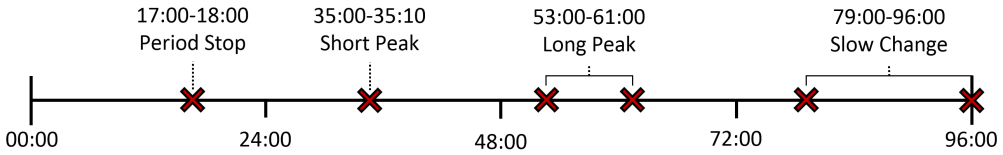

Figure 6.2: Timeline of the attacks.

case there should be no alarms for any log line types or time steps. For this scenario, the final user is assumed to be an intruder who gained unauthorized access to the system after a social engineering attack. The frequencies of performed actions by this user do not cohere to the overall behavior of the others. Over the course of the simulation, the attacker performs the following anomalous actions:

1. **Missing periodic event**: After 17 hours, the intruder blocks the automatized program for 1 hour from performing the scheduled event. The log lines corresponding to the planned actions in this time window are therefore absent from the log file. Afterwards, the program continues to work as usual.

2. **Sudden frequency peak**: After 35 hours, the attacker only clicks on a specific button for 10 minutes. This produces a peak in the recorded frequency of the corresponding log line types.

3. **Long-term frequency increase**: After 53 hours, the intruder clicks on another button for the following 8 hours. This produces a plateau in the recorded frequency of the corresponding log line types.

4. **Gradual frequency increase**: After 79 hours, the attacker clicks on a third button until the end of the simulation. It is assumed for this case that the attacker knows about the installed anomaly detection system and therefore tries to outsmart the algorithm by changing the behavior gradually. This avoids rapid changes in frequency that trigger alarms, while at the same time the learning effect of the algorithm adapts to the malicious behavior. The future prediction intervals are influenced by this behavior and after some time, the attacker is able to further increase the clicking frequency. By continuing this pattern for a sufficient duration, the attacker should be able to inject arbitrary large frequency changes.

Figure 6.2 shows these attacks on a timeline. Large gaps of several hours were intentionally left between the injections in order to ensure that previous attacks do not affect the likelihood of a future attack being detected.

In total, the generated log file consists of around 4 million log lines. The average length of the log lines is around 246 characters in the raw form and around 218 characters after removing consecutive white spaces during the preprocessing stage. More than 99.7% of the preprocessed log lines have a length below 600 characters while the longest line has a length of 72862 characters.

6.2 Evaluation Environment

The log data was generated on a general purpose workstation, with an Intel Xeon CPU E5-1620 v2 at 3.70 GHz 8 cores and 16 GB memory, running Ubuntu 16.04 LTS operating system. The workstation runs virtual servers for an Apache Web server hosting the MANTIS Bug Tracker System, a MySQL database and a reverse proxy. The log messages of these systems are aggregated using syslog.

The anomaly detection algorithm runs on a 64-bit Windows 7 machine, with an Intel i7-3770 CPU at 3.4 GHz and 8 GB memory. The algorithm was implemented in Java version 1.8.0.141.

6.3 Results

The evaluation results of an anomaly detection system are frequently dependent on several factors. The log data at hand often requires fine-tuning of parameters that influence the effectiveness and efficiency of the algorithm. Settings that optimize the result on a specific data set may perform poorly on another.

Therefore, several parameter settings are tested and compared. This is supposed to show the influence of the selected parameters on relevant characteristics, including the quality of the results and the runtime. Insights gained by such experiments are expected to generalize also on other data sets and aid the identification of appropriate parameter ranges in practical applications. Important parameters and their default values are as follows:

- Similarity threshold t: The threshold used for the incremental generation of the static cluster maps within each time window. A higher threshold means that log lines must be more similar in order to be grouped within the same cluster. This also means that a higher threshold usually relates to a higher total amount of clusters. Unless otherwise stated, $t = 0.9$.

- Overlap thresholds θ and θ_{part}: The thresholds used within the transition detection algorithm. A higher threshold θ means that clusters from different cluster maps require a higher overlap in order to be connected. Furthermore, θ_{part} specifies the minimum overlap that is required for clusters that contribute to a transition, i.e., to be part of a merge or split. Unless otherwise stated, $\theta = 0.7$ and $\theta_{part} = 0.2$.

- Time window size t_w: The cluster maps are generated within each time window. A larger time window size therefore means that more log lines are used for each cluster map. Unless otherwise stated, $t_w = 15$ minutes.

- Prediction level α: The prediction level influences the boundaries that are used for determining whether a point is an anomaly or not. The boundaries form a tube around the cluster feature and the prediction level specifies the thickness of that

tube. A higher prediction level leads to a smaller size of the tube and therefore increases the amount of detected anomalies. Unless otherwise stated, $\alpha = 0.01$, i.e., $1 - \alpha = 99\%$ of the non-anomalous data points should be located within the boundaries.

6.3.1 Operability

The introduced clustering model and the anomaly detection mechanism are designed to only focus on dynamic changes that occur over multiple time windows rather than other forms of anomalies that occur only in a single time window. Such other anomalies are for example outliers, i.e., log lines that form their own cluster in the construction phase due to their high dissimilarity to all the other lines and are also not allocated to clusters from other time windows during the allocation phase. Clearly, there is no way to identify any temporal changes from such lines as they simply do not exhibit any dynamic features.

While outliers are an extreme example, also clusters containing more than 1 element and existing for several time steps cannot always be used for detection. Due to the fact that the ARIMA model requires a number of historic data points before the prediction interval is reasonably initialized, only anomalies detected in clusters that have been existing for at least 5 time steps are considered. This is necessary to avoid the relatively high amount of false alarms that occur in the first few time steps impairing the evaluation results.

There may therefore be a concern that only few log lines remain that are eventually contained in the cluster evolution process. Such a situation would indicate a low credibility and could also lead to a poor performance of the algorithm due to the fact that most of the log lines are never considered for the anomaly detection procedure.

It is therefore important to understand the factors that influence the ability of forming permanent and stable clusters that exist for at least the minimum amount of time steps required for a proper anomaly detection. For a given data set, the functioning of the clustering model in combination with the overlap coefficient determines whether clusters are effectively mapped over time. The most relevant parameter is thus the similarity threshold t used in the clustering process. Figure 6.3 shows the amount of log lines that formed or contributed to evolving clusters which exist for at least 5 time steps plotted against t. It can clearly be seen that low thresholds ($t \leq 0.5$) cause that only $20\% - 30\%$ of the total amount of log lines end up in evolving clusters while large thresholds (especially $0.8 \leq t \leq 0.9$) achieve a representation of more than 90% of all log lines. There is thus a clear preference for larger thresholds.

The reason behind this tendency is as follows. Lower thresholds lead to fewer clusters in each cluster map as well as wider ranges of log line types being grouped into the same clusters. Due to the fact that there is always only one cluster representative responsible for representing all the contained log lines, also largely dissimilar log lines are represented by this initial line as only a small similarity is required. However, in other time steps it is likely that very different cluster representatives are selected. This is because most of the representatives are selected among the first few log lines occurring after the start of

a new time window. This behavior is independent from the threshold and caused by the fact that many of these initial log lines do not match any of the representatives of the few clusters existing at this point, thereby forming their own clusters. As there is no proper ordering of log lines but many different types of log lines are grouped within the same cluster, the selection of the cluster representatives is more or less random where obviously more frequently occurring log lines are more likely to end up as representatives. In other words, low similarity thresholds cause that the cluster representatives are not appropriately representing the log lines allocated to this cluster.

The line types allocated to the different set of representatives also do not properly correspond to any cluster that exists in another time step. When it comes to the allocation phase, the log lines that established a cluster in one time step are therefore likely to be allocated to several clusters from another time step and no clear connections between single clusters are made. Due to the fact that the overlap metric measures the strength of these connections, the minimum thresholds θ and θ_{part} for establishing transitions are not reached. Without transitions, no cluster evolution takes place and hence there remain large clusters in every time step that do not have any correspondences in the preceding or succeeding time windows and are thus unable to contribute to anomaly detection. Reducing θ or θ_{part} would increase the percentage of log lines represented by evolving clusters, but is also not a reasonable solution due to the fact that rather dissimilar clusters would be connected and the number of splits and merges would inappropriately increase without actually representing the development of the system behavior.

On the other hand, large thresholds lead to the formation of many clusters, with most of them containing highly similar log lines. This corresponds to a finer granularity of clustering. In every time window, the cluster maps consist of clusters with similar representatives and thus log lines from one cluster are correctly allocated to a specific cluster in another time step during the allocation phase. Thereby, distinct connections between clusters are established and the overlap metric successfully creates transitions between the cluster maps. Furthermore, it can be assumed that the developments of single clusters actually represent specific log line types, while random splits and merges are kept at a minimum so that any changes are clear and comprehensible.

6.3.2 Cluster Evolution Visualizations

Visualizations of the time-series retrieved by cluster evolution techniques aid the understanding of the anomaly detection mechanism and show the functionality of the introduced approach. Only taking evolving clusters that were tracked for at least 20 time steps into account, over 300 such clusters were found. In the following, the plots of some interesting clusters that exhibit representative features are shown.

There are log lines that appear more frequently than others, e.g., every set of SQL queries belonging to a certain action always start with a log line stating “Init DB”. All of the injected attacks involve the creation of SQL queries, therefore the cluster corresponding to this line type is expected to display the effects of all attacks. Figure 6.4

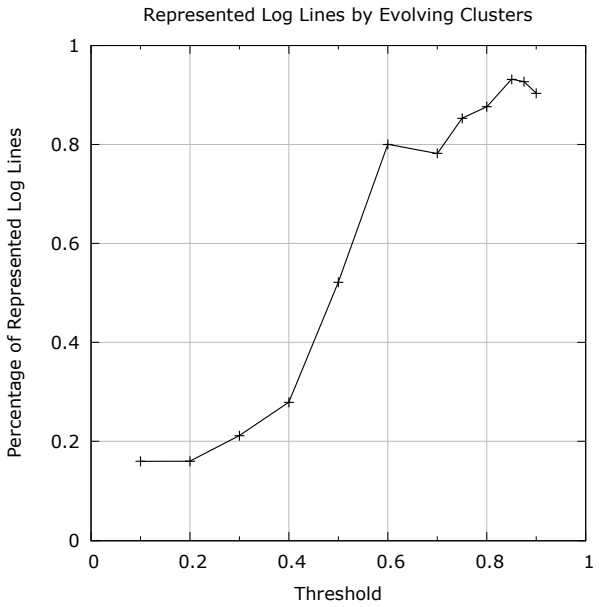

Figure 6.3: Effectiveness of cluster evolution approach evaluated by the relative amount of log lines that are represented by an evolving cluster that exists for at least 5 time steps.

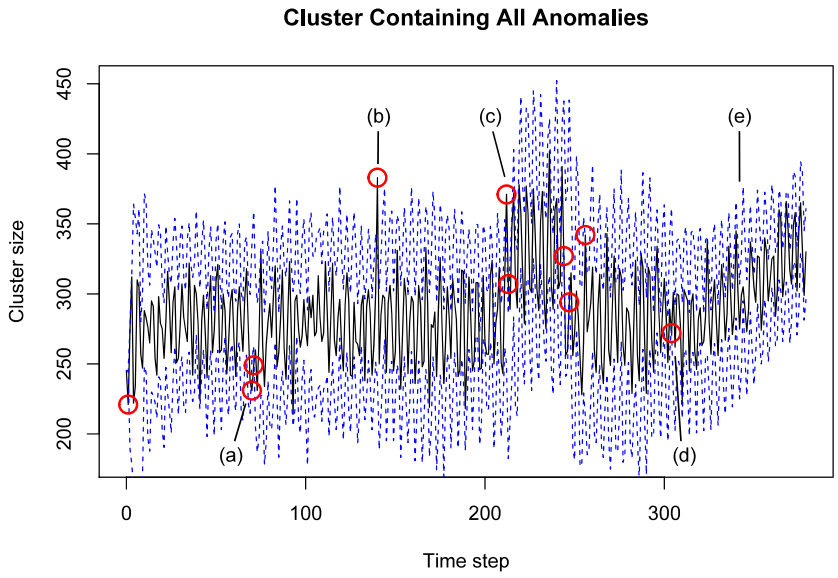

Figure 6.4: Development of cluster corresponding to log line “Init DB”. Solid black line: Actual measured cluster size. Dashed blue line: One-step ahead prediction boundaries. Red circles: Detected anomalies.

shows the development of the size of this cluster over time. Note that each time step covers a 15 minute period of occurring log lines, thus the total amount of 384 time steps corresponds to a time span of 96 hours. The actual measured cluster size (solid black line) is approximated in every step in order to predict the boundaries (dashed blue lines) for the following step. Whenever the actual size in the next step falls outside of the tube that is formed around the curve, an anomaly is detected and marked with a red circle. The figure shows the following features:

(a) A correctly detected anomaly, i.e., a true positive. This anomaly is corresponding to the missing periodic event attack. The figure shows that the periodic behavior is captured very well throughout the simulation as the position of the prediction interval corresponds to the up-and-down movements of the cluster size. The time-series model learns the correct period in less than 10 time steps and is further able to keep the correct periodicity while adjusting to outliers (b), level shifts (c) and changes in trend (e).

(b) Another correctly detected anomaly corresponding to the sudden increase in frequency. Due to the fact that the duration of this attack is smaller than the length of a time window, only one time step is affected.

(c) Another correctly detected anomaly corresponding to the start of the long-term increase in frequency. Also the decrease of frequency at the end of the plateau is detected correctly. The figure clearly shows that it only takes very few time steps until the time-series model adapts to the new mean value as there are no anomalies detected in between the start and the end of the plateau. This demonstrates the self-learning ability of the time-series model to adapt to changing environments without the need to manually interfere.

(d) An incorrectly detected anomaly, i.e, a false positive.

(e) An undetected anomaly, i.e., a false negative. This anomaly corresponds to the gradual frequency increase. As expected, this anomaly is not detected by the time-series model due to the fact that the frequency change is not rapid enough in any time step so that the actual cluster size would fall outside of the tube.

All the other points that are not detected as anomalies are therefore correctly undetected data points, i.e., true negatives. This visual evaluation already suggests that the introduced anomaly detection methodology is successfully able to retrieve the temporal development of log line frequencies and identify irregular behavior and rapid changes while keeping the amount of false classifications at a minimum.

One of the main advantages of this anomaly detection methodology is that a high number of evolving clusters is retrieved from the log file. Thereby, each cluster development may exhibit some specific characteristics that would remain unnoticed when only considering the log file as a whole. Some interesting cluster size developments corresponding to

specific log line types are considered in the following. All of them describe differently complex SQL SELECT statements.

Figure 6.5 shows a cluster that does not have a periodic component, i.e., the log line representing this cluster is not part of the set of log lines that is created by the event that is periodically triggered. Therefore, the curve appears smoother and the prediction limits do not show any regular up-and-down movements.

The figure shows very well that the ARIMA model initially requires a number of time steps until stable and appropriately sized prediction intervals are computed. After around 20 time steps, the tube flows around the cluster size with a constant width and no single false positive anomaly is detected in the first 200 time steps. Clearly, neither the missing periodic event anomaly nor the short-term frequency peak anomaly are detected in this cluster size development due to the fact that the log lines generated by these events are not contained in this evolving cluster.

The specificity of this cluster leads to a lower average amount of contained log lines which makes it easier to detect the remaining anomalies. Although Fig. 6.4 already indicated those anomalies as well, the magnitude of the change was rather low, i.e., the average cluster size in each time step increased only by 25% from around 280 to 350 when the long-term frequency increase anomaly occurred. The cluster size displayed in Fig. 6.5 however increases by 400% from around 20 to 80. Furthermore, this plot shows several anomalies detected during the gradual frequency increase anomaly while no anomalies were detected in the previous plot. Especially in the case where a higher prediction level (i.e., a larger thickness of the tube) is used, anomalies possibly remain undetected in larger clusters as their contributions to the overall sizes vanish compared to the totality of contributions from other log lines, while they become clearly visible in smaller clusters. It should now be apparent that it is necessary to consider all cluster size evolutions for anomaly detection in order to ensure that anomalies which manifest themselves only in very specific clusters are detected as well.

Figure 6.6 shows the development of a cluster size that mostly contains periodically occurring log lines. Due to the fact that these lines appear very regularly and there are no other log lines causing noise or other fluctuations, the ARIMA model is able to approximate the curve very closely. This precise fitting of the prediction interval also leads to several false positives that appear in the later time steps. As expected, several anomalies are detected in the time step where the missing periodic event anomaly occurs. The high frequency of the cycle compared to the rather long overall runtime makes it difficult to see the influence of the anomaly on the cluster size in detail. Therefore, Fig. 6.7 shows the relevant segment of the time-series that contains the missing periodic event around time step 70. The figure also shows that just one missing event is not sufficient for the ARIMA model to unlearn the periodic pattern so that once the curve has returned to its normal behavior the tube is already in an appropriate shape. Obviously, the ARIMA model would also adjust to permanent changes in periodicity after several time steps have passed.

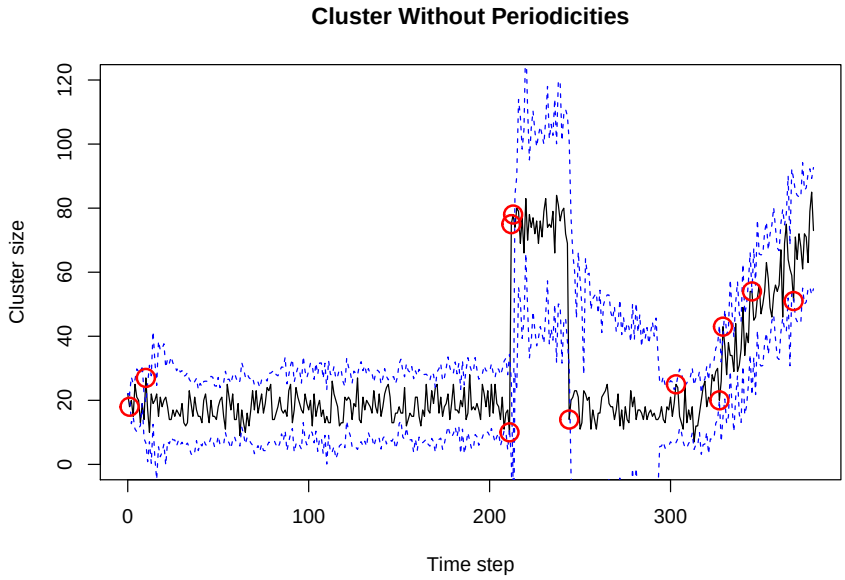

Figure 6.5: Development of a cluster size that corresponds to log lines affected by anomalies regarding long-term frequency increase and the gradual frequency increase.

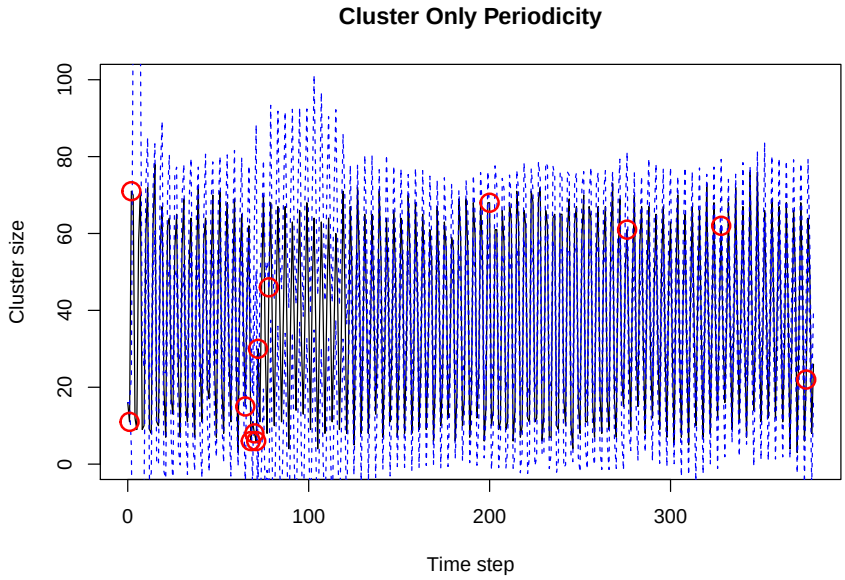

Figure 6.6: Development of a cluster size that corresponds to periodically occurring log lines. This allows the detection of the missing periodic event anomaly.

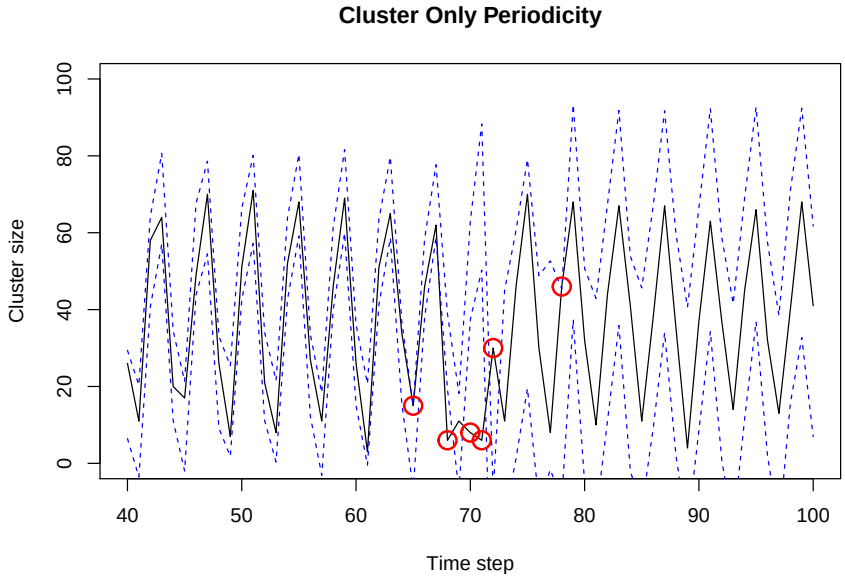

Figure 6.7: Detailed view on the segment where the missing periodic event anomaly occurs.

Finally, Fig. 6.8 shows the size corresponding to a cluster that specifically contains the log lines affected by the short-term frequency peak anomaly. The fact that this anomaly manifests itself distinctly in the plot is due to the same reasons already mentioned for the long-term frequency increase. This visualization also shows the influence of an anomaly on the following forecasts of the prediction interval. As it can be seen, the interval increases to an unexpectedly large thickness for around 50 time steps after the anomaly occurs. This is the result of the large error computed between the anomalous value and the estimated value increasing the range according to Eq. (5.18).

It was already mentioned before that anomalies occurring within a certain amount of time steps after an attack are more unlikely to be detected and it would be an intuitive idea to remove the errors generated by anomalies from the computations in order to avoid these adverse effects. This would also include the errors generated by false positives. However, these errors are essential for computing a correct prediction interval. For example, it would be unjustified to remove the error from the false positives occurring around time step 100 from the computations as this would result in too narrow prediction limits.

In an unsupervised setting there is no way for the algorithm to determine whether the detected anomaly is a false positive or a true positive. Given that false positives usually outnumber true positives it is therefore recommended to keep all the errors for the computations, despite their influence on future predictions. In practical applications however, a reasonable compromise would be to omit the errors from anomalous values that have been confirmed as actual anomalies by a human system administrator.

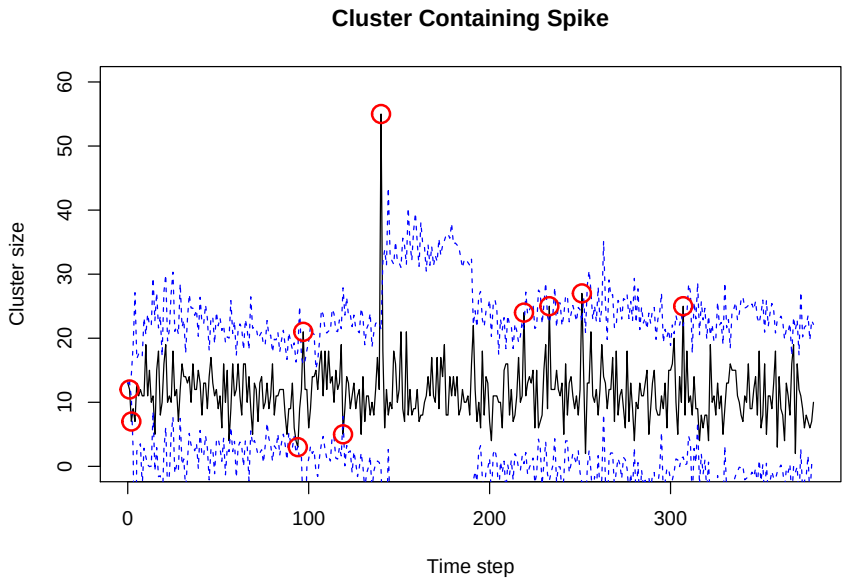

Figure 6.8: Development of a cluster size that shows the short-term frequency peak anomaly.

6.3.3 Rates

Quantitative metrics are required for an appropriate comparison of results achieved by different parameter settings or different anomaly detection algorithms. Due to the fact that the data set is known to be free of anomalies except for the four injected attacks, it is possible to compute measures that assess the quality of the result in an objective and replicative way. As already mentioned, a ground truth table containing both the time steps and samples of log lines that were generated during the respective attacks was assembled. An anomaly is detected by the algorithm at a specific detection time step t_d and for a specific cluster with representative r_d. Anomalies are only counted as true positives (TP) if the ground truth table contains an entry with expected time step $t_e \in [t_d - 30min, t_d + 60min]$ and expected log line content r_e so that $s_{Lev}(r_e, r_d) \geq t$, i.e., the similarity must be greater or equal to the threshold that was used for clustering. Detected anomalies that do not fulfill one of these requirements are counted as false positives (FP). Entries from the ground truth table that remain undetected are counted as false negatives (FN), i.e., actually occurring anomalies that remained undetected. Due to the fact that the ground truth table only contains anomalies but does not define all the non-anomalous log lines and their respective time steps, the amount of true negatives (TN) cannot simply be counted and has to be determined computationally. TN is therefore computed by summing up all the time steps in every cluster that were not detected as anomalies and subtract FN. Table 6.1 gives an overview about the relationships of these values in a so-called confusion matrix.

		Actual State	
		Anomalous	Normal
Detected State	Anomalous	TP	FP
	Normal	FN	TN

Table 6.1: Confusion matrix

It is common to compute rates based on these measures (Powers, 2008). The true positive rate (TPR), also known as Recall (R) or sensitivity, represents the fraction of correctly detected anomalies from the total amount of actually existing anomalies and is computed by

$$TPR = R = \frac{TP}{TP + FN} \tag{6.1}$$

Clearly, $TPR \in [0, 1]$ and a high TPR is favorable as it implies that many of the actually existing anomalies have been detected. However, the TPR neglects the amount of FP and is therefore on its own not an appropriate measure for determining the overall quality of the detection.

In order to overcome this problem, the false positive rate (FPR) is frequently used in combination with the TPR. The FPR represents the fraction of incorrectly detected anomalies from the total amount of non-anomalous data points. The rate is computed by

$$FPR = \frac{FP}{FP + TN} \tag{6.2}$$

and again $FPR \in [0, 1]$. Anomaly detection techniques try to keep the amount of false alarms at a minimum, hence a lower FPR is favorable.

A related metric that is often mentioned in combination with R is the Precision (P) or confidence that measures the fraction of correctly detected anomalies from all the detected anomalies. The Precision is computed by

$$P = \frac{TP}{TP + FP} \tag{6.3}$$

and again $P \in [0, 1]$. A higher P is favorable as it implies a lower amount of FP in relation to the amount of TP.

6.3.4 ROC Analysis

The previously mentioned metrics are now used for creating the Receiver-Operator-Characteristic (ROC). ROC plots are frequently used for comparing classifiers in machine learning (Powers, 2008). In the ROC analysis, TPR on the y-axis is plotted against FPR on the x-axis. Thereby, each classifier with a specific setting yields exactly one point in the ROC plot. In general, the classifier that results in a point that is closest to the top-left corner of the plot is seen as the best classifier. This is due to the fact

that the top-left corner of the plot represents $TPR = 1$ and $FPR = 0$ and thus the most favorable result. Furthermore, the first median ($TPR = FPR$) shows the performance of a random guesser. Any proper classifier should therefore yield a point which lies left to the first median.

Clearly, different parameter settings affect TPR as well as FPR and therefore influence the point that is yielded in the ROC plot. In order to visualize the influence of a parameter, it may be desirable to connect these points with a line. The curve formed by these lines should start at the bottom-left corner of the plot ($TPR = 0$, $FPR = 0$) representing a very weak threshold where no anomalies are detected and end up at the top-right corner of the plot ($TPR = 1$, $FPR = 1$) representing a very strict threshold that leads to all data points being detected as anomalous. However, not all parameters are fitted for this purpose. It is a requirement that with an increasing value for this parameter, both TPR and FPR do not decrease, i.e., the resulting curve in the ROC plot must always be directed towards the top-right corner of the plot. Such a parameter is the prediction level α. A low value for α leads to a large prediction interval and therefore only anomalies with extreme deviations are detected, i.e., the algorithm will miss most of the anomalies but also exhibit a low false alarm rate ($TPR \approx 0$, $FPR \approx 0$). On the other hand, a high value for α leads to a small prediction interval which will lead to almost all data points being detected as anomalies, independent from whether they actually are anomalies or not ($TPR \approx 1$, $FPR \approx 1$). Somewhere in between lies a trade-off value that maximizes TPR and minimizes FPR. In the following, α is varied from very low to very high values in order to compute the points necessary to plot the curves and to identify this optimal value.

Choosing the value for the similarity threshold t is a crucial decision. Figure 6.9 shows the ROC plot with a number of curves that represent the quality of the result for different settings of t. Thresholds smaller than 0.5 were omitted as they do not represent a sufficient amount of log lines so that their TPR and FPR are not necessarily valid for other attacks or datasets. The displayed curves all lie left to the first median, thereby showing that the algorithm is successfully able to detect anomalies in the correct clusters and the correct time steps. Comparing the performances for different thresholds with each other does not give a distinct trend as both large ($t = 0.9$) and small ($t = 0.5$) thresholds are outperformed by $t = 0.85$ and $t = 0.875$. This mostly corresponds to the insights gained in Section 6.3.1 as these thresholds also achieved the highest percentage of log lines contained in evolving clusters. It can therefore be concluded that for an appropriately good performance the selected threshold must fit the data well, i.e., be large enough to correctly differentiate the occurring log line types while still being small enough to avoid the creation of outliers due to IDs, time stamps or other artifacts in the strings.

The previously mentioned trade-off between a high TPR and a low FPR can also be seen in every curve. In practice, α should be set so that the resulting point lies at the "bend" of the curve that is close to the top-left corner. It must be specifically noted at this point that $TPR = 1$ would indicate that an anomaly was detected at the correct

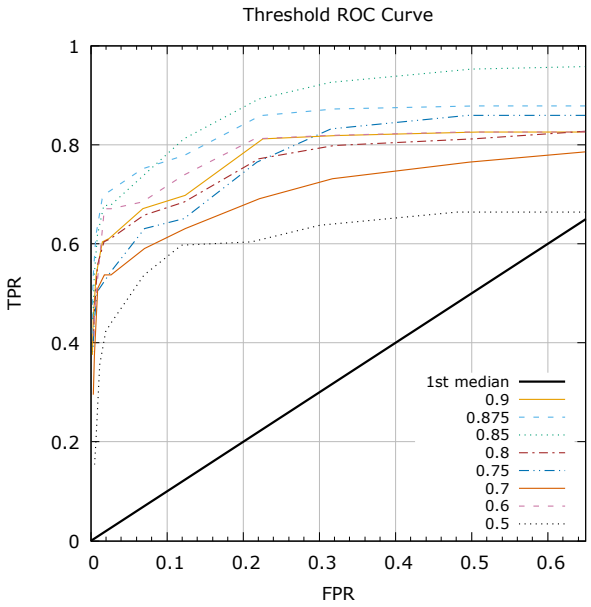

Figure 6.9: ROC curves showing anomaly detection performance for different similarity thresholds.

time step in every single cluster that is related to the action performed during the attack. As false alarms are often regarded as adverse and it would already be sufficient to detect every anomaly in at least one cluster in order to count the corresponding anomaly as detected, it should not be the main priority to achieve the highest possible TPR but rather to reasonably minimize FPR. E.g., for $t = 0.875$ a practically reasonable α would be 0.001 as it achieves $TPR = 0.618$ and $FPR = 0.007$.

Similar to the threshold, the time window size t_w is difficult to chose due to the fact that an appropriate setting requires experiments or knowledge about the data. A small time window size corresponds to a fine granularity, i.e., changes that occur in short periods are more distinctly present. However, this also causes that there may be time windows where certain types of infrequently appearing log lines are not present at all, which would produce a cut in the time-series of an evolving cluster. On the other hand, large window sizes have the advantage of producing less volatile time-series but also average out short-term anomalies which are then less likely detected. Furthermore, large time window sizes increase the detection time, i.e., the duration between an attack occurring and this attack being detected. This is due to the fact that on average an attack occurs uniformly distributed within any time window, however is only detected at the end of that time window. The average detection time is therefore computed by $\frac{t_w}{2}$.

The anomalies occurring in the data set used in this scenario are differently affected by the choice of t_w. Both long-term attacks are less likely to remain undetected due to the fact that their caused change in system behavior usually exceeds a reasonably sized

time window. On the other hand, the short-term frequency increase only takes place during 10 minutes and may therefore be missed when large window sizes are used, e.g., the anomalous lines make up $\frac{2}{3}$ of the time window when $t_w = 15$ minutes but only make up $\frac{1}{6}$ of the time window when $t_w = 60$ minutes. Furthermore, the time window size may emphasize, distort or hide periodic behavior occurring in the data due to the fact that the time window and the periodic interval are misaligned or the time window size exceeds the period altogether. It is therefore difficult to predict how the selection of the time window size influences the ability of detecting attacks related to the periodic behavior.

In order to investigate the influence of the data set, ROC plots were also produced from the results of the anomaly detection processing a more complex data set. For the generation of this data set, identical attacks were scheduled, i.e., the affected time intervals and the absolute number of anomalous log lines are the same as before. However, 5 additional users constantly produce log lines corresponding to normal behavior, thereby increasing the total number of log lines that are clustered in every time window. This causes the effects of the anomalies to appear smaller due to the fact that there is a smaller fraction of anomalous lines in every time window and hence it is more likely that the algorithm misses actual anomalies.

Figure 6.10 shows the comparison between the ROC curves from the complex data set displayed as solid lines and the previously computed ROC curves displayed as dashed lines. As it could be expected, the curves from the complex data set mostly lie below the original curves due to the previously mentioned issues, indicating a decline in performance. Surprisingly, $t = 0.7$ poses an exception to this pattern as the results improved on the complex data set. The reason for this is that only for this similarity threshold the percentage of log lines contained in evolving clusters increased compared to the simple log file, meaning that more evolving clusters could be used for performing anomaly detection. This issue was already mentioned in Section 6.3.1 and is linked to the fact that even though there are more users producing noise, the overall variability of the cluster sizes recorded at the end of each time window decreases and thus better predictions can be made. The decreased variability also means that it is less likely that the development of an evolving cluster is interrupted due to randomly occurring non-representative log lines within certain time windows.

Figure 6.11 shows the ROC curve for different settings of the time window size t_w. Again, all curves lie left of the first median. However, some of the curves perform worse than others. Especially for a reasonably low false positive rate, e.g., $FPR = 0.03$, the small time window sizes < 60 minutes clearly outperform the larger time window sizes. This is due to the previously mentioned issues that appear when the time window size is larger than the duration of the attacks. Furthermore, $t_w = 7$ minutes causes a similarly poor performance due to the fact that it does not properly align with the periodic behavior that repeats every 60 minutes and thereby leads to highly sporadic cluster developments in all clusters that are affected by the corresponding log lines. The best possible choices that aim at a low FPR are therefore time windows that are factors of this interval, e.g., 15 minutes or 30 minutes.

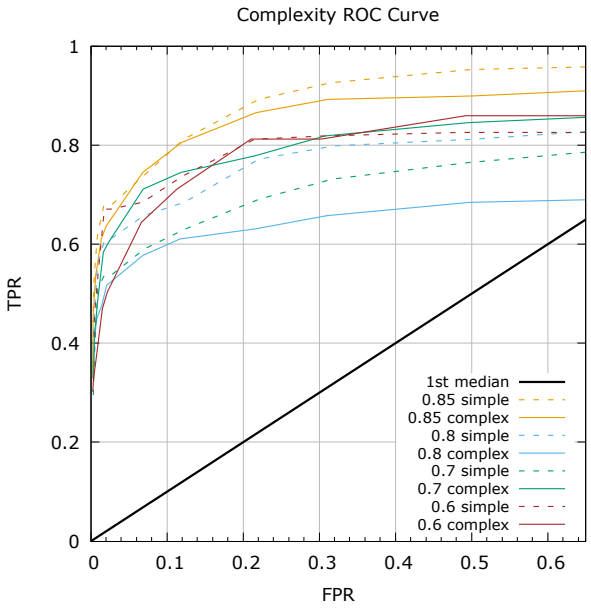

Figure 6.10: ROC curves showing the influence of data complexity on the anomaly detection performance.

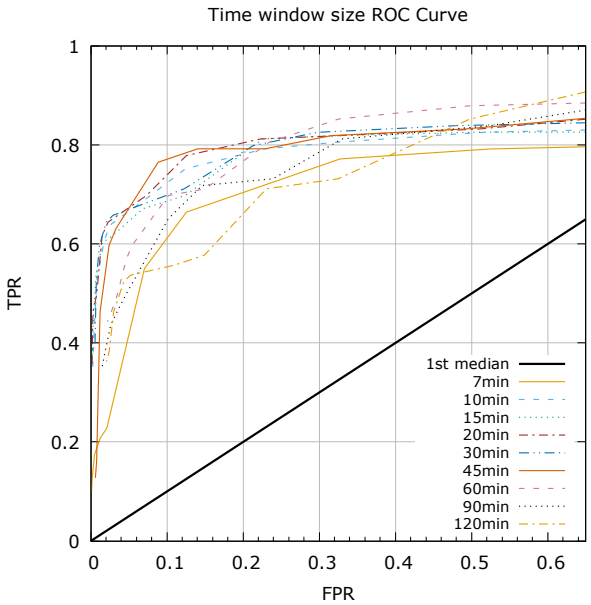

Figure 6.11: ROC curves showing anomaly detection performance for different time window sizes.

6.3.5 Precision-Recall Plots

In the scenario used for this evaluation the amount of non-anomalous data points largely outweighs the amount of anomalous points and the data set is thus strongly imbalanced. This is assumed to be generally valid for real-world log files and is a requirement for anomaly detection based on unsupervised learning due to the fact that the algorithm cannot differentiate between normal and abnormal behavior otherwise. However, Saito and Rehmsmeier (2015) point out that ROC plots are often misleading whenever evaluation is carried out on such imbalanced data sets. This is due to the fact that the precision P is left out from the evaluation, even though it is usually an easy to interpret value and an important criterion when ranking and selecting classifiers. Furthermore, the first median that represents a baseline in the ROC plot does not change depending on the balancedness of the data sets. The authors therefore recommend precision-recall (PR) plots that use P rather than FPR and are able to dynamically adjust the baseline.

Figure 6.12 shows the PR plot for different thresholds. The TPR that was plotted on the y-axis in the ROC curve is now plotted as recall R on the x-axis. Classifiers that achieve a high recall ($R = 1$) and a high precision ($P = 1$) are favorable and yield points that are close to the top-right corner of the plot. The selection of the prediction level shows a similar trade-off that was already visible in the ROC plot: A large prediction interval caused by a low α leads to a high P but only a low R. This is due to the fact that only few data points are detected as anomalous, because most deviations do not exceed the prediction limits, while the ones that are detected are very likely actual anomalies. Resulting points therefore lie closer to the top-left corner of the plot. On the other hand, a small prediction interval caused by a high α leads to a high R but only a low P due to the fact that random noise is frequently misclassified as an anomaly. Again, reasonable values that maximize both P and R are achieved by moderate prediction levels.

The baseline for the precision that represents the performance of a random guesser is determined by the fraction of actual anomalies defined in the ground truth table from the total amount of data points. However, due to the fact that the total amount of data points changes when the threshold is changed, this baseline is different for every curve. Furthermore, the average precision by a random guesser would only marginally exceed 0, because the total amount of data point heavily exceeds the amount of entries in the ground truth table. For these reasons, the baselines were omitted from the plot.

The curves in the plot indeed give a different view of the results. Especially $t = 0.6$ yields $P = 0.28$ and $R = 0.67$ for $\alpha = 0.005$ and thereby outperforms all other thresholds for that α. Besides that, the threshold values $t = 0.875$ and $t = 0.85$ that have already performed well in the ROC plot are again superior to all the other thresholds over a broader range of the prediction level.

Figure 6.13 shows the PR plot for different time window sizes. A surprising observation is that the highest P is reached by $t_w = 20$ minutes. Nevertheless, $t_w = 30$ minutes appears to be the best choice as it maximizes both P and R. The bad performances of time window sizes that exceed 60 minutes that were already visible in the ROC plot are

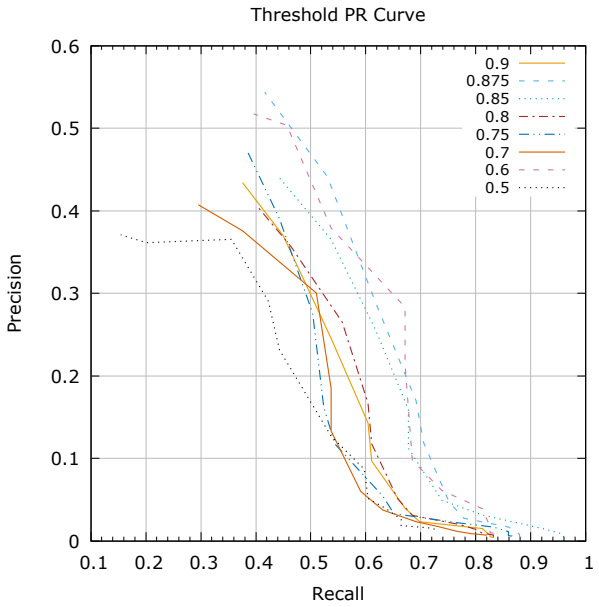

Figure 6.12: Precision-recall plot showing anomaly detection performance for different similarity thresholds.

confirmed again. The PR plot however shows much clearer that $t_w = 7$ minutes is in no way able to keep up with the other thresholds regarding the precision. Interestingly, the precision of $t_w = 45$ minutes contradicts the general trend and shows a declining precision for low α values. The reason for this is that with an increasing prediction interval, more correctly classified true positives fall inside the tube and are not reported as anomalies anymore.

6.3.6 Runtime and Scalability

In the previous sections, the influences of the similarity threshold t and time window size t_w on the quality of the result were investigated and it became apparent that specific ranges of values are clearly superior to others regarding TPR, FPR or P. It would therefore be natural to select the values of the parameters solely on the performance visible in the ROC or PR plot. However, anomaly detection is computationally intensive and therefore also the runtime of the algorithm must be considered when choosing parameter values.

Figure 6.14 shows the recorded runtime for different threshold values. From a runtime perspective, moderate values around $t = 0.6$ are clearly favorable over extreme values close to 0 or 1. The reason for this lies in the functional interaction of two main components of the anomaly detection algorithm: The clustering and the fitting of ARIMA models. First of all, it should be clear that the extreme case of $t = 0$ is not reasonable as this would cause all log lines to be allocated into a single cluster, which is identical to skipping the

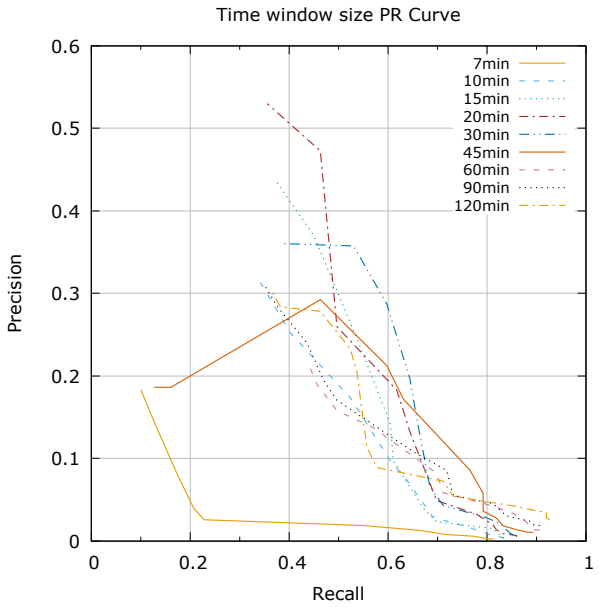

Figure 6.13: Precision-recall plot showing anomaly detection performance for different time window sizes.

clustering at all and just observing the total amount of log lines occurring in every time window. In general, a smaller threshold results in less clusters and it is therefore logical that less time needs to be spent with approximating and extrapolating ARIMA models. Furthermore, there is a smaller number of clusters available that a line could be allocated to. However, due to the small similarity threshold, most of these clusters are potential candidates due to the fact that neither filtering for line length nor filtering according to n-grams effectively eliminates cluster candidates. Therefore, the computationally complex string distance metric has to be computed multiple times for every incoming log line in order to determine the best fitting cluster. With increasing t, these filtering steps become more effective and therefore the runtime decreases initially.

However, an increasing number of clusters also shows its effects once the critical point around $t = 0.6$ is passed. Each evolving cluster corresponds to a time-series that needs to be represented as an ARIMA model which is then used for prediction. This is a time-consuming operation that dominates the runtime for larger thresholds. Especially from $t = 0.8$ on the runtime grows at a very high rate and it was not possible to perform any runs with $t > 0.9$ due to limitations in available processing power. Finally, setting $t = 1$ is most probably not a reasonable choice as only completely identical lines are grouped in clusters. Especially for data sets where all log lines are unique due to time stamps or IDs, this would mean that no cluster evolution takes place. Although this would reduce the runtime due to the fact that it is not necessary to create any ARIMA models, there is obviously no way to detect dynamic anomalies with this setting.

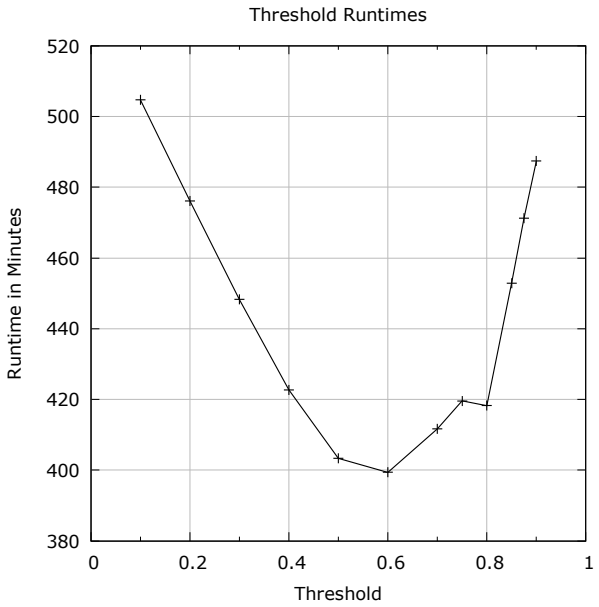

Figure 6.14: Total runtimes for different similarity thresholds.

The importance of the ability for online processing was emphasized multiple times throughout this thesis. Although a reasonably sized runtime indicates that the computational requirements lie within a manageable scope, efficiency is only half of the story. In order to ensure the ability to process data streams of arbitrary length, a linear scalability is required, i.e., the runtime must only linearly depend on the number of log lines. In other words, increasing the amount of log lines by a factor n should only increase the required runtime by a factor smaller or equal to n. This characteristic was empirically verified by measuring the elapsed time after each set of $50,000$ log lines. Figure 6.15 shows these cumulated times for different similarity thresholds that were retrieved when performing anomaly detection on the scenario used in this evaluation. This means that also the attacks are contained in the log data, however no significant changes in the runtime can be observed at any point. As it can be seen, the runtimes exhibit a linear behavior independent from the chosen threshold. The cumulated runtimes retrieved after all log lines have been processed directly correspond to the total runtimes shown in the previous plot and thus identical conclusions about the influence of the threshold on the required runtime can be drawn.

Next, the influence of the time window size on the runtime is investigated. Figure 6.16 shows the total runtimes that were measured for different values of t_w. It can immediately be seen that there is an increase in runtime for large time windows ($t_w > 60$ minutes), while $t_w = 30$ minutes and $t_w = 15$ minutes show the lowest runtimes. Other than for the threshold, the time required for fitting the ARIMA model is the major cause for the different runtimes, while the time used for clustering is more or less independent from t_w.

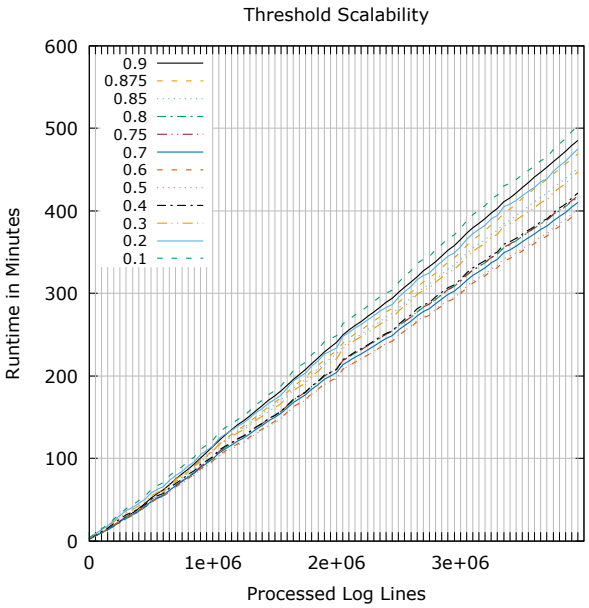

Figure 6.15: Plot showing the continuously measured runtime that is required for processing a certain amount of log lines. The runtime scales linearly for all considered similarity thresholds.

The reason for this is that always a fixed number of data points from the past are stored, independent from the selected time window size. For example, when 48 historic values are stored for fitting the time-series model, the last 48 hours are used for computing the prediction intervals when $t_w = 60$ minutes, but only 12 hours are used when $t_w = 15$ minutes. A smaller time window size therefore causes the algorithm to remove data from the past earlier and anomalies in the historic data thus have less influence on the predictions. It is also computationally faster to fit an ARIMA model on the anomaly-free data, thereby leading to shorter runtimes for smaller time window sizes. It was already mentioned that t_w values that do not align with the periodic behavior of the data cause the resulting curves to be more spurious. For example, the periodic interval in this scenario consists of two consecutive 30 minute intervals and can therefore not be captured properly with $t_w = 20$ minutes, but rather $t_w = 15$ minutes or $t_w = 30$ minutes.

An alternative implementation could only store data points that lie within a fixed duration in the past, e.g. the last 24 hours. This has some obvious implications regarding the runtime. The smaller the time window size is selected, the more data points have to be considered when fitting the ARIMA model. Therefore, the runtime is likely to increase for short time windows and decrease for large time windows in this setting.

Finally, Figure 6.17 shows the scalability of the algorithm with respect to several time window sizes. The interpretation is identical to the previous scalability plot that focused on the threshold. Again, all curves exhibit a linear behavior and it can thus be concluded

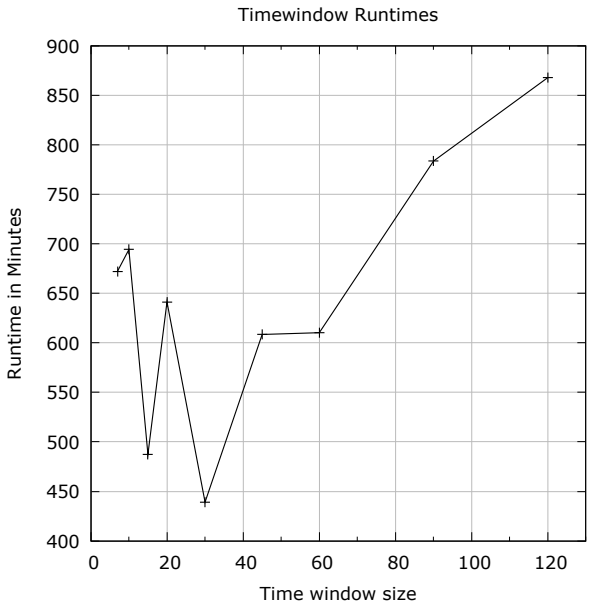

Figure 6.16: Total runtimes for different time window sizes.

that the algorithm is able to ensure the processing of streams in real time, independent from the selection of t_w and under the assumption that the available computation power is sufficient to process incoming log lines faster than they appear.

6.4 Aggregated Detection

Aggregating the detected outliers from all clusters gives an overview of the current state of the whole system. The anomaly score introduced in Eq. (5.34) is a measure for the deviation from the expected cluster sizes from all clusters that exist for at least 20 time steps. Figure 6.18 shows the anomaly scores yielded in every time step using $t = 0.875$, $t_w = 15$ minutes and $\alpha = 0.00001$. The value for α was chosen rather small in order to minimize the influence of false positives while emphasizing the large deviations occurring in specific clusters where the anomalies manifest themselves clearly. The intervals shaded red indicate the appearances and durations of the injected anomalies.

The plot confirms the previous observations regarding the successful detection of the first three anomalies. The anomaly relating to the long-term frequency increase once more shows very distinctly that the algorithm only detects changes of the system behavior, but immediately adjusts to shifts and trends. For that reason, almost no anomalies are detected within the interval shaded red. Only when the system returns to the normal behavior, the anomaly score again increases to an alarming level. There further exist a few spikes outside of the shaded regions which are either false positives or artifacts from previous anomalies, e.g., the small spikes around time step 250.

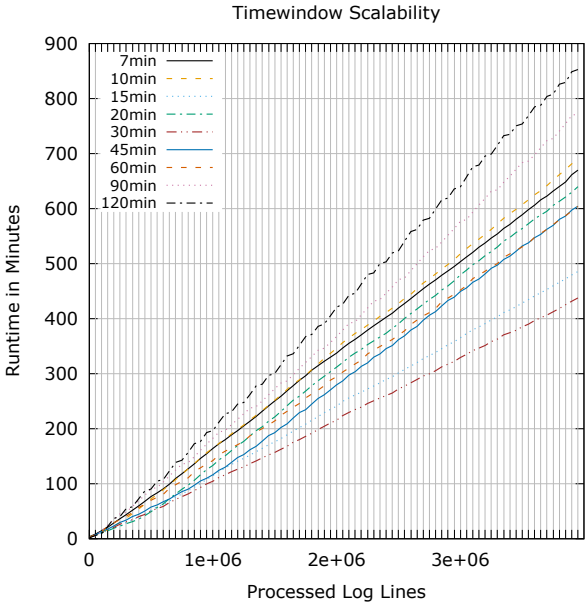

Figure 6.17: Plot showing the continuously measured runtime that is required for processing a certain amount of log lines. The runtime scales linearly for all considered time window sizes.

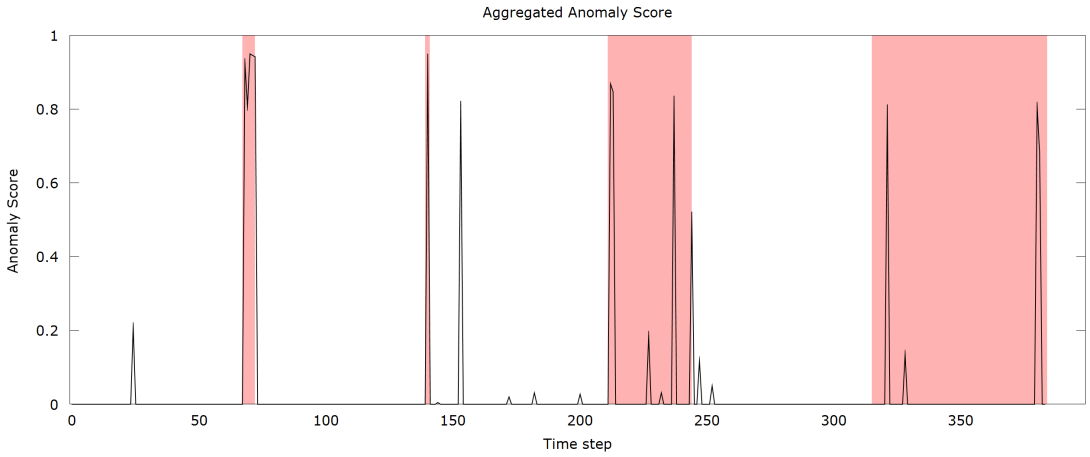

Figure 6.18: Anomaly score of every time step. The phases of occurring attacks are shaded in red.

In practice there may be a need for an alarm threshold that defines at what level the aggregated anomaly score triggers a warning. In the case of this scenario, a threshold around 0.4 would be reasonable as all the anomalies and only a single false positive would raise alarms. Other settings such as a higher α however require a different threshold in order to compensate that anomalies are detected more easily. Moreover, the intensity of anomalous effects may change depending on the dataset and application area. Selecting an appropriate alarm threshold thus remains an open question and should be decided individually for each system by observation and empirical knowledge. Due to the fact that the anomaly score again creates a time-series, it may stand to reason to employ time-series analysis methods for detecting anomalous time steps. As periodic behavior is already captured by the models used to approximate the developments of the individual clusters and the anomaly score remains at 0 over long periods, one-step ahead prediction using ARIMA does not appear to be an appropriate modeling technique. Most probably, filters that involve outlier detection are a fitting choice for this task. A theoretical investigation of such a filter is given in Section 5.4, but due to the wide-spread prominence and diversity of these methods, no evaluation of such filters was carried out in the scope of this thesis.

6.5 Application on Real Log Data

As previously mentioned, real log data does not allow a proper evaluation because of a missing ground truth table. Nevertheless, processing a real dataset gives additional insights into the applicability of the methodology. Therefore, the anomaly detection algorithm was applied using log data that was collected within the Austrian Institute of Technology (AIT). Both automatized processes that operate with different periodicities as well as erratic human behavior contribute to the captured logs. Due to data security reasons, no details or samples of this dataset are given.

The logs were recorded over the course of 1 week without any interruptions. The following parameters were selected: $t = 0.8$, $t_w = 30$ minutes, $\theta = 0.7$ and $\theta_{part} = 0.2$. With this setting, more than 90% of the total amount of log lines are successfully represented by evolving clusters that exist for at least 5 time steps. Figure 6.19 shows an exemplary cluster size development that exhibits interesting characteristics. A time window size of 30 minutes means that 1 day is represented by 48 time steps. The patterns that are visible in the plot appear accordingly to this interval on the first, second, third and seventh day. As expected, these plateaus are correctly identified as anomalies. The ARIMA model was set up to recognize periodicities repeating within a maximum of 12 hours and this pattern is therefore not learned. In order to capture larger periodicities, it is recommended to run the algorithm multiple times in parallel but with different time window sizes. This avoids that too many historic data points have to be stored in memory.

Other plots exhibited artifacts that corresponded to the displayed cluster size but differed in shape and magnitude. Furthermore, some clusters captured the highly precise periodic behavior of scheduled programs or the noisy behavior of randomly interfering events.

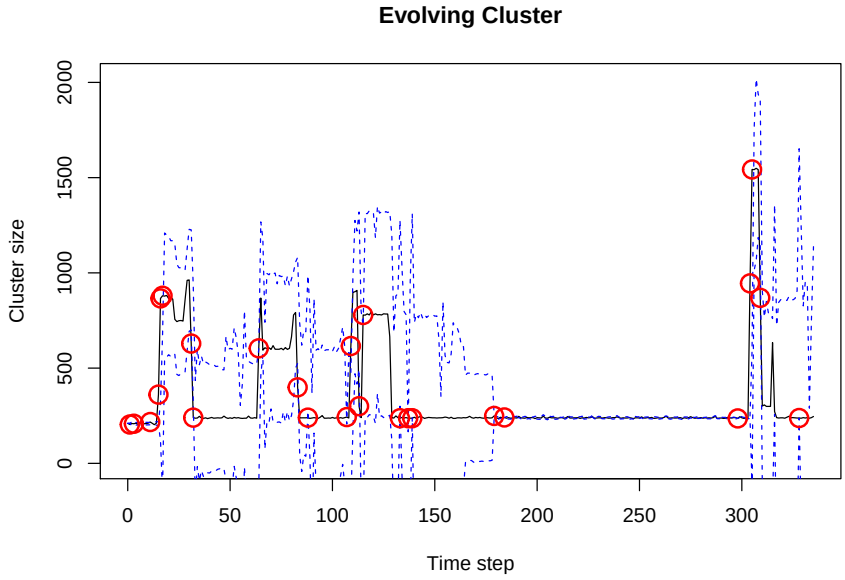

Figure 6.19: Development of a cluster size measured on real data.

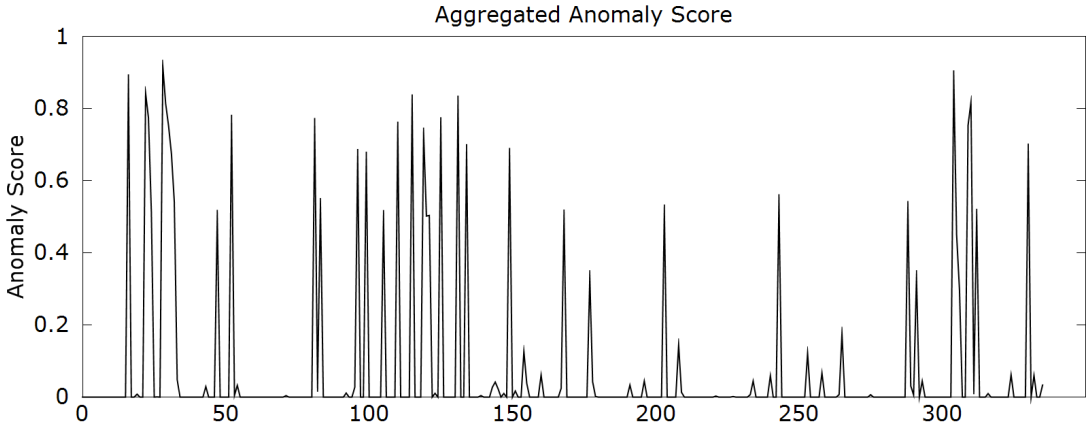

Figure 6.20: Anomaly score computed on real data.

Figure 6.20 shows the aggregated anomaly score computed for all clusters that exist for at least a total of 100 time steps. Due to the fact that the artifacts displayed in the previous plot are of rather high magnitude and also appeared in other clusters, they are also visible in the anomaly score plot. Furthermore, anomalies from other clusters also yield a rather high anomaly score, e.g., at time step 150. It appears that the most anomalies occurred during the third day and only few anomalies occurred on the fourth, fifth and sixth day.

Obviously there was no known attack taking place during the time where the log was captured and the detected anomalies only correspond to harmless events such as updates. It is not surprising that the algorithm detects such events as anomalies due to the fact that the attacks are assumed to manifest themselves in exactly the same way. There is thus no simple way for an algorithm to differentiate between an anomaly corresponding to an attack and an anomaly caused by regular events such as updates. This is an obvious drawback that affects unsupervised self-learning anomaly detection methods in general. Nevertheless, in critical systems the risk of frequent false alarms is accepted in order to ensure that attacks that are difficult to detect for other methods are not overseen. Furthermore, the knowledge of the human administrator about scheduled events that are possibly detected as anomalies should be sufficient to dismiss many of the false positives immediately. All in all, both the evaluation on semi-synthetic data as well as the results retrieved from real data suggest the effectiveness of the introduced anomaly detection methodology.

CHAPTER 7

Conclusion and Future Work

A methodology for dynamic anomaly detection in log files was introduced in this thesis. This methodology comprises a sequence of steps that are carried out for every processed log line. Within a time window of predefined length, all log lines occurring within that window are incrementally grouped by similarity in order to establish a static cluster map. Furthermore, the log lines are also allocated to the already existing cluster maps that were created to the preceding and succeeding time windows. For any two neighboring cluster maps in the resulting sequence, this clustering scheme establishes a connection between individual clusters of two static cluster maps that previously did not share any common elements. An overlap metric then measures the likelihood of a cluster from one cluster map transforming into another cluster from the succeeding cluster map.

An algorithm then allows the detection of advanced transitions such as splits or merges. In addition, metrics that give information about the state of the individual clusters, their relationships and interdependencies were defined. These metrics as well as cluster features such as the size conveniently form time-series that are effectively approximated by ARIMA models. A one-step ahead forecast of these models in every time step then allows the detection of anomalies in a fast and reliable manner. Due to an efficient clustering algorithm and a time-series model that only requires a fixed set of preceding values, this methodology allows online processing of data streams in real-world applications.

In order to demonstrate the applicability, a prototype that incorporates the anomaly detection methodology was placed into an illustrative scenario. Within this scenario, a semi-synthetically generated log file was used for evaluating the ability of detecting certain types of known anomalies. Furthermore, the influence of important parameters as well as the effects on the runtime and scalability of the algorithm were investigated. Promising performances were achieved for most parameter values and fine-adjustments could optimize the quality of the results. The anomaly detection applied on the developments of individual clusters as well as an anomaly score aggregated over all evolving clusters showed clear peaks when the injected attacks affected the system behavior. When

the algorithm was executed on a real log file, several relevant changes of system behavior that corresponded to alterations of log frequencies could be observed.

The parameters that were changed in the evaluation obviously only make up for a small part of the potential modifications that could positively influence the ability of detecting certain types of anomalies or processing log files of a different composition. For example, the incremental clustering algorithm could be replaced by any other machine learning technique that is able to group similar strings in an unsupervised manner as long as it is possible to separate a construction and an allocation phase that are required for determining the evolution of the clusters. Despite higher computational requirements, the mentioned overlap metric that takes multiple time windows into account when determining the connections between clusters could result in more reliable evolutions.

Even more possibilities for enhancements exist for approximating time-series other than ARIMA models that may be beneficial due to their robustness or ability to pick up periodicities, e.g., Holt-Winters or ETS. Alternatively, filters that are usually applied for smoothing time-series are also suited for detecting outliers on historic data without any need for computing prediction intervals of future values. Such a robust filter was discussed in Section 5.4. Moreover, methods that employ change point analysis are promising solutions to detect anomalies that cause gradual and long-term changes. Especially the fourth injected anomaly that remained undetected in many clusters could successfully be identified by such techniques (Killick et al., 2012).

The evaluation focused on the size of the cluster as it directly represents the frequency of the corresponding log line types in the respective time windows and was thus appropriate to detect the injected attacks. However, many other useful metrics that were defined in Sections 4.2 and 4.3 were left out from practical analysis, even though they could be better fitted for special types of anomalies. Other than performing anomaly detection on time-series created on each of these features alone there is also the possibility to combine them and apply multivariate outlier detection as it was described in Section 5.5. This could enhance the ability of the algorithm to identify otherwise undetected attacks.

A more fundamental problem arises when tracking individual clusters at split or merge points. Several reasonable solutions were stated in the thesis, however all of them lead to cutting away all but one evolving cluster from the plot displaying the development of a certain feature. Rather than focusing only on one of the paths, a sophisticated technique could be able to analyze the graph of evolving clusters as a whole. Both the detection of anomalies in this graph as well as a visualization of the overall cluster developments pose interesting research topics.

Finally, the problem of rather high amounts of false positives that all anomaly detection techniques suffer from remains unsolved. It appears that every attempt to make the algorithm more robust against such influences at the same time restricts its ability of detecting certain types of anomalies. Trustworthy and up-to-date domain knowledge about the specific use case would be required in order to additionally support the self-learning methods in differentiating between normal behavior and an actual anomaly.

List of Figures

Bibliography

L. Akoglu, H. Tong, and D. Koutra. Graph-based anomaly detection and description: A survey. *CoRR*, abs/1404.4679, 2014. URL http://arxiv.org/abs/1404.4679.

N. B. Amor, S. Benferhat, and Z. Elouedi. Naive bayes vs decision trees in intrusion detection systems. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, SAC '04, pages 420–424, New York, NY, USA, 2004. ACM. ISBN 1-58113-812-1. doi: 10.1145/967900.967989. URL http://doi.acm.org/10.1145/967900.967989.

J. Andreasson and C. Geijer. Log-based anomaly detection for system surveillance. Master's thesis, 2015.

N. Andrienko, G. Andrienko, and P. Gatalsky. Exploratory spatio-temporal visualization: an analytical review. *Journal of Visual Languages & Computing*, 14(6):503–541, 2003.

S. Asur, S. Parthasarathy, and D. Ucar. An event-based framework for characterizing the evolutionary behavior of interaction graphs. *ACM Trans. Knowl. Discov. Data*, 3(4):16:1–16:36, Dec. 2009. ISSN 1556-4681. doi: 10.1145/1631162.1631164. URL http://doi.acm.org/10.1145/1631162.1631164.

C. C. Bilgin and B. Yener. Dynamic network evolution: Models, clustering, anomaly detection. *IEEE Networks*, 2006.

D. Brauckhoff, K. Salamatian, and M. May. Applying pca for traffic anomaly detection: Problems and solutions. In *INFOCOM 2009, IEEE*, pages 2866–2870. IEEE, 2009.

J. Breier and J. Branišová. *Anomaly Detection from Log Files Using Data Mining Techniques*, pages 449–457. Springer, Berlin, Heidelberg, 2015. ISBN 978-3-662-46578-3. doi: 10.1007/978-3-662-46578-3_53. URL http://dx.doi.org/10.1007/978-3-662-46578-3_53.

P. Bródka, S. Saganowski, and P. Kazienko. Ged: the method for group evolution discovery in social networks. *Social Network Analysis and Mining*, 3(1):1–14, Mar 2013. ISSN 1869-5469. doi: 10.1007/s13278-012-0058-8. URL http://dx.doi.org/10.1007/s13278-012-0058-8.

A. Cardenas, S. Amin, B. Sinopoli, A. Giani, A. Perrig, and S. Sastry. Challenges for securing cyber physical systems. In *Workshop on future directions in cyber-physical systems security*, volume 5, 2009.

A. Carmi, F. Septier, and S. J. Godsill. The Gaussian mixture mcmc particle algorithm for dynamic cluster tracking. In *2009 12th International Conference on Information Fusion*, pages 1179–1186, July 2009.

D. Chakrabarti, R. Kumar, and A. Tomkins. Evolutionary clustering. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 554–560, New York, NY, USA, 2006. ACM. ISBN 1-59593-339-5. doi: 10.1145/1150402.1150467. URL http://doi.acm.org/10.1145/1150402.1150467.

J. Chan, J. Bailey, and C. Leckie. Discovering correlated spatio-temporal changes in evolving graphs. *Knowl. Inf. Syst.*, 16(1):53–96, July 2008. ISSN 0219-1377. doi: 10.1007/s10115-007-0117-z. URL http://dx.doi.org/10.1007/s10115-007-0117-z.

V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009. ISSN 0360-0300. doi: 10.1145/1541880.1541882. URL http://doi.acm.org/10.1145/1541880.1541882.

Y. Chi, X. Song, D. Zhou, K. Hino, and B. L. Tseng. On evolutionary spectral clustering. *ACM Trans. Knowl. Discov. Data*, 3(4):17:1–17:30, Dec. 2009. ISSN 1556-4681. doi: 10.1145/1631162.1631165. URL http://doi.acm.org/10.1145/1631162.1631165.

S. C. Chin, A. Ray, and V. Rajagopalan. Symbolic time series analysis for anomaly detection: A comparative evaluation. *Signal Process.*, 85(9):1859–1868, Sept. 2005. ISSN 0165-1684. doi: 10.1016/j.sigpro.2005.03.014. URL http://dx.doi.org/10.1016/j.sigpro.2005.03.014.

P. Cortez, M. Rio, M. Rocha, and P. Sousa. Multi-scale internet traffic forecasting using neural networks and time series methods. *Expert Systems*, 29(2):143–155, 2012. ISSN 1468-0394. doi: 10.1111/j.1468-0394.2010.00568.x. URL http://dx.doi.org/10.1111/j.1468-0394.2010.00568.x.

J. Cryer and K. Chan. *Time Series Analysis: With Applications in R*. Springer Texts in Statistics. Springer New York, 2008. ISBN 9780387759593. URL https://books.google.at/books?id=bHke2k-QYP4C.

P. Esling and C. Agon. Time-series data mining. *ACM Comput. Surv.*, 45(1):12:1–12:34, Dec. 2012. ISSN 0360-0300. doi: 10.1145/2379776.2379788. URL http://doi.acm.org/10.1145/2379776.2379788.

T. Falkowski, J. Bartelheimer, and M. Spiliopoulou. Mining and visualizing the evolution of subgroups in social networks. In *2006 IEEE/WIC/ACM International Conference*

on Web Intelligence (WI 2006 Main Conference Proceedings)(WI'06), pages 52–58, Dec 2006. doi: 10.1109/WI.2006.118.

P. Filzmoser, R. Maronna, and M. Werner. Outlier identification in high dimensions. *Comput. Stat. Data Anal.*, 52(3):1694–1711, Jan. 2008. ISSN 0167-9473. doi: 10.1016/j.csda.2007.05.018. URL http://dx.doi.org/10.1016/j.csda.2007.05.018.

U. Fiore, F. Palmieri, A. Castiglione, and A. De Santis. Network anomaly detection with the restricted boltzmann machine. *Neurocomput.*, 122:13–23, Dec. 2013. ISSN 0925-2312. doi: 10.1016/j.neucom.2012.11.050. URL http://dx.doi.org/10.1016/j.neucom.2012.11.050.

R. Fried. Robust filtering of time series with trends. *Journal of Nonparametric Statistics*, 16(3-4):313–328, 2004. doi: 10.1080/10485250410001656444. URL http://dx.doi.org/10.1080/10485250410001656444.

Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 149–158, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3895-2. doi: 10.1109/ICDM.2009.60. URL http://dx.doi.org/10.1109/ICDM.2009.60.

M. Ghodsi, B. Liu, and M. Pop. Dnaclust: accurate and efficient clustering of phylogenetic marker genes. *BMC Bioinformatics*, 12, 2011.

J. Goh, S. Adepu, M. Tan, and Z. S. Lee. Anomaly detection in cyber physical systems using recurrent neural networks. In *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, pages 140–145, Jan 2017. doi: 10.1109/HASE.2017.36.

M. K. Goldberg, M. Hayvanovych, and M. Magdon-Ismail. Measuring similarity between sets of overlapping clusters. In *Proceedings of the 2010 IEEE Second International Conference on Social Computing*, SOCIALCOM '10, pages 303–308, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4211-9. doi: 10.1109/SocialCom.2010.50. URL http://dx.doi.org/10.1109/SocialCom.2010.50.

M. Goldstein and S. Uchida. A comparative evaluation of unsupervised anomaly detection algorithms for multivariate data. *PLOS ONE*, 11(4):1–31, 04 2016. doi: 10.1371/journal.pone.0152173. URL https://doi.org/10.1371/journal.pone.0152173.

D. Greene and P. Cunningham. Multi-view clustering for mining heterogeneous social network data. In *Paper presented at the Workshop on Information Retrieval over Social Networks, 31st European Conference on Information Retrieval (ECIR'09), Toulouse, France, April 6-9*, 2009.

D. Greene, D. Doyle, and P. Cunningham. Tracking the evolution of communities in dynamic social networks. In *2010 International Conference on Advances in Social*

Networks Analysis and Mining, pages 176–183, Aug 2010. doi: 10.1109/ASONAM.2010.17.

M. Gupta, J. Gao, C. Aggarwal, and J. Han. *Outlier Detection for Temporal Data*. Morgan & Claypool Publishers, 2014. ISBN 1627053751, 9781627053754.

S. He, J. Zhu, P. He, and M. R. Lyu. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 207–218, Oct 2016. doi: 10.1109/ISSRE.2016.21.

D. J. Hill and B. S. Minsker. Anomaly detection in streaming environmental sensor data: A data-driven modeling approach. *Environ. Model. Softw.*, 25(9):1014–1022, Sept. 2010. ISSN 1364-8152. doi: 10.1016/j.envsoft.2009.08.010. URL http://dx.doi.org/10.1016/j.envsoft.2009.08.010.

Y. Huang, B. Niu, Y. Gao, L. Fu, and W. Li. Cd-hit suite: a web server for clustering and comparing biological sequences. *Bioinformatics*, 26(5):680–682, 2010. doi: 10.1093/bioinformatics/btq003. URL +http://dx.doi.org/10.1093/bioinformatics/btq003.

R. J. Hyndman. The difference between prediction intervals and confidence intervals. https://robjhyndman.com/hyndsight/intervals/, 2013. [Online; accessed 07-August-2017].

H. Hyyrö. A bit-vector algorithm for computing levenshtein and damerau edit distances. *Nordic J. of Computing*, 10(1):29–39, Mar. 2003. ISSN 1236-6064. URL http://dl.acm.org/citation.cfm?id=846090.846095.

C. S. Jensen, D. Lin, and B. C. Ooi. Continuous clustering of moving objects. *IEEE Transactions on Knowledge and Data Engineering*, 19(9):1161–1174, Sept 2007. ISSN 1041-4347. doi: 10.1109/TKDE.2007.1054.

A. Juvonen, T. Sipola, and T. Hämäläinen. Online anomaly detection using dimensionality reduction techniques for http log analysis. *Computer Networks*, 91:46 – 56, 2015. ISSN 1389-1286. doi: http://dx.doi.org/10.1016/j.comnet.2015.07.019. URL http://www.sciencedirect.com/science/article/pii/S1389128615002650.

K. Kent and M. P. Souppaya. Sp 800-92. guide to computer security log management. Technical report, Gaithersburg, MD, United States, 2006.

M. Khalilian and N. Mustapha. Data stream clustering: Challenges and issues. *CoRR*, abs/1006.5261, 2010. URL http://arxiv.org/abs/1006.5261.

R. Killick, P. Fearnhead, and I. A. Eckley. Optimal detection of changepoints with a linear computational cost. *Journal of the American Statistical Association*, 107(500):1590–1598, 2012.

G. Kondrak. *N-Gram Similarity and Distance*, pages 115–126. Springer, Berlin, Heidelberg, 2005. ISBN 978-3-540-32241-2. doi: 10.1007/11575832_13. URL https://doi.org/10.1007/11575832_13.

S. Krishnamurthy, S. Sarkar, and A. Tewari. Scalable anomaly detection and isolation in cyber-physical systems using bayesian networks. In *Dynamic Systems and Control Conference*, 2014. ISBN 978-0-7918-4619-3. doi: 10.1115/DSCC2014-6365. URL http://dx.doi.org/10.1115/DSCC2014-6365.

C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 251–261, New York, NY, USA, 2003. ACM. ISBN 1-58113-738-9. doi: 10.1145/948109.948144. URL http://doi.acm.org/10.1145/948109.948144.

P. Lee, L. V. S. Lakshmanan, and E. E. Milios. Incremental cluster evolution tracking from highly dynamic network data. In *2014 IEEE 30th International Conference on Data Engineering*, pages 3–14, March 2014. doi: 10.1109/ICDE.2014.6816635.

M. Li, X. Chen, X. Li, B. Ma, and P. M. B. Vitanyi. The similarity metric. *IEEE Transactions on Information Theory*, 50(12):3250–3264, Dec 2004. ISSN 0018-9448. doi: 10.1109/TIT.2004.838101.

W. Li, L. Jaroszewski, and A. Godzik. Clustering of highly homologous sequences to reduce the size of large protein databases. *Bioinformatics*, 17 3:282–3, 2001.

W. Li, L. Jaroszewski, and A. Godzik. Sequence clustering strategies improve remote homology recognitions while reducing search times. *Protein Engineering, Design and Selection*, 15(8):643–649, 2002. doi: 10.1093/protein/15.8.643. URL +http://dx.doi.org/10.1093/protein/15.8.643.

H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24, 2013.

E. Lughofer and M. Sayed-Mouchaweh. Autonomous data stream clustering implementing split-and-merge concepts – towards a plug-and-play approach. *Information Sciences*, 304:54 – 79, 2015. ISSN 0020-0255. doi: http://dx.doi.org/10.1016/j.ins.2015.01.010. URL http://www.sciencedirect.com/science/article/pii/S0020025515000328.

W. J. Masek and M. S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18 – 31, 1980. ISSN 0022-0000. doi: http://dx.doi.org/10.1016/0022-0000(80)90002-1. URL http://www.sciencedirect.com/science/article/pii/0022000080900021.

V. Metsis, I. Androutsopoulos, and G. Paliouras. Spam filtering with naive Bayes-which naive Bayes? In *CEAS*, volume 17, pages 28–69, 2006.

R. Mitchell and I.-R. Chen. A survey of intrusion detection techniques for cyber-physical systems. *ACM Comput. Surv.*, 46(4):55:1–55:29, Mar. 2014. ISSN 0360-0300. doi: 10.1145/2542049. URL http://doi.acm.org/10.1145/2542049.

C. C. Noble and D. J. Cook. Graph-based anomaly detection. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 631–636, New York, NY, USA, 2003. ACM. ISBN 1-58113-737-0. doi: 10.1145/956750.956831. URL http://doi.acm.org/10.1145/956750.956831.

B. Pincombe. Anomaly detection in time series of graphs using arma processes. *Asor Bulletin*, 24(4):2, 2005.

D. Powers. Evaluation: From precision, recall and F-factor to ROC, informedness, markedness and correlation. 2, 01 2008.

M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1039834.1039864.

H. Rosling and Z. Zhang. Health advocacy with gapminder animated statistics. *Journal of Epidemiology and Global Health*, 1(1):11 – 14, 2011. ISSN 2210-6006. doi: http://dx.doi.org/10.1016/j.jegh.2011.07.001. URL http://www.sciencedirect.com/science/article/pii/S2210600611000074.

J. P. Rouillard. Refereed papers: Real-time log file analysis using the simple event correlator (sec). In *Proceedings of the 18th USENIX Conference on System Administration*, LISA '04, pages 133–150, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1052676.1052694.

T. Saito and M. Rehmsmeier. The precision-recall plot is more informative than the roc plot when evaluating binary classifiers on imbalanced datasets. *PLOS ONE*, 10 (3):1–21, 03 2015. doi: 10.1371/journal.pone.0118432. URL https://doi.org/10.1371/journal.pone.0118432.

K. A. Scarfone and P. M. Mell. Sp 800-94. guide to intrusion detection and prevention systems (idps). Technical report, Gaithersburg, MD, United States, 2007.

J. A. Silva, E. R. Faria, R. C. Barros, E. R. Hruschka, A. C. P. L. F. d. Carvalho, and J. a. Gama. Data stream clustering: A survey. *ACM Comput. Surv.*, 46(1): 13:1–13:31, July 2013. ISSN 0360-0300. doi: 10.1145/2522968.2522981. URL http://doi.acm.org/10.1145/2522968.2522981.

F. Skopik, G. Settanni, R. Fiedler, and I. Friedberg. Semi-synthetic data set generation for security software evaluation. In *2014 Twelfth Annual International Conference on Privacy, Security and Trust*, pages 156–163, July 2014. doi: 10.1109/PST.2014.6890935.

A. Sperotto, R. Sadre, and A. Pras. Anomaly characterization in flow-based traffic time series. In *Proceedings of the 8th IEEE International Workshop on IP Operations and Management*, IPOM '08, pages 15–27, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87356-3. doi: 10.1007/978-3-540-87357-0_2. URL http://dx.doi.org/10.1007/978-3-540-87357-0_2.

M. Spiliopoulou, I. Ntoutsi, Y. Theodoridis, and R. Schult. Monic: Modeling and monitoring cluster transitions. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 706–711, New York, NY, USA, 2006. ACM. ISBN 1-59593-339-5. doi: 10.1145/1150402.1150491. URL http://doi.acm.org/10.1145/1150402.1150491.

L. Spitzner. Honeypots: Catching the insider threat. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pages 170–179. IEEE, 2003.

J. Stearley. Towards informatic analysis of syslogs. In *Cluster Computing, 2004 IEEE International Conference on*, pages 309–318. IEEE, 2004.

M. Takaffoli, F. Sangi, J. Fagnan, and O. R. Zäıane. Community evolution mining in dynamic social networks. *Procedia-Social and Behavioral Sciences*, 22:49–58, 2011.

M. Thottan and C. Ji. Anomaly detection in ip networks. *IEEE Transactions on signal processing*, 51(8):2191–2204, 2003.

M. Toyoda and M. Kitsuregawa. Extracting evolution of web communities from a series of web archives. In *Proceedings of the Fourteenth ACM Conference on Hypertext and Hypermedia*, HYPERTEXT '03, pages 28–37, New York, NY, USA, 2003. ACM. ISBN 1-58113-704-4. doi: 10.1145/900051.900059. URL http://doi.acm.org/10.1145/900051.900059.

R. Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations Management (IPOM 2003) (IEEE Cat. No.03EX764)*, pages 119–126, Oct 2003. doi: 10.1109/IPOM.2003.1251233.

C. Vehlow, F. Beck, P. Auwärter, and D. Weiskopf. Visualizing the evolution of communities in dynamic graphs. *Computer Graphics Forum*, 34(1):277–288, 2015. ISSN 1467-8659. doi: 10.1111/cgf.12512. URL http://dx.doi.org/10.1111/cgf.12512.

B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962. doi: 10.1080/00401706.1962.10490022. URL http://amstat.tandfonline.com/doi/abs/10.1080/00401706.1962.10490022.

M. Wurzenberger, F. Skopik, M. Landauer, P. Greitbauer, R. Fiedler, and W. Kastner. Incremental clustering for semi-supervised anomaly detection applied on log data. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, page 31. ACM, 2017.

K. S. Xu, M. Kliger, and A. O. Hero Iii. Adaptive evolutionary clustering. *Data Mining and Knowledge Discovery*, 28(2):304–336, 2014.

W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 117–132, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629587. URL http://doi.acm.org/10.1145/1629575.1629587.

W. Yassin, N. I. Udzir, Z. Muda, and M. N. Sulaiman. K-means clustering and naive Bayes classification for intrusion detection. In *Proceedings of the 4th International Conference on Computing and Informatics*, 2013. URL http://publisher.unimas.my/ojs/index.php/JITA/article/view/45.

M. Yu, G. Li, D. Deng, and J. Feng. String similarity search and join: a survey. *Frontiers of Computer Science*, 10(3):399–417, Jun 2016. ISSN 2095-2236. doi: 10.1007/s11704-015-5900-5. URL https://doi.org/10.1007/s11704-015-5900-5.

A. Zhou, F. Cao, W. Qian, and C. Jin. Tracking clusters in evolving data streams over sliding windows. *Knowl. Inf. Syst.*, 15(2):181–214, May 2008. ISSN 0219-1377. doi: 10.1007/s10115-007-0070-x. URL http://dx.doi.org/10.1007/s10115-007-0070-x.