View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Oulu Repository - Jultika

Developing a Log File Analysis Tool: A Machine Learning Approach for Anomaly Detection

University of Oulu
Faculty of Information Technology and
Electrical Engineering / Information
Processing Science
Master's Thesis
Tapio Anttila
3.6.2020

Abstract

Log files, which record information about all events during the execution of a software, are important in troubleshooting tasks. However, modern software systems produce large quantities of complex logs, and their manual inspection is laborious and time-consuming. Therefore, technologies such as machine learning have been used to automate log file analysis. Anomaly detection is an especially popular approach, since anomalies in the log files are typically caused by erroneous behaviour of the software.

In this study, open source data mining and machine learning solutions are utilized to process log files collected from devices running embedded Linux. Following the Design Science Research methodology, a Python program called *sgologs* is developed. The tool uses components from *logparser* and *loglizer* toolkits to pre-process the input log file, train an unsupervised machine learning model, and detect anomalies on the input file.

The *loglizer* tools have not been used with Linux logs in previous research, possibly because they are rather difficult for automated processing. This finding is verified in this study as well, as the measured anomaly detection accuracy scores are quite modest. Nevertheless, *sgologs* is able to detect anomalies in the log files, with swift processing times, at least when certain things are taken into consideration. If the user is aware of these factors, *sgologs* can definitely point towards real anomalies in the Linux log files. Thus, the tool could be used in real-life settings to simplify debugging tasks, whenever logs are used as a source of information.

Keywords
Log file analysis, Linux log files, anomaly detection, machine learning, unsupervised learning, log parsing, design science research

Supervisor
University lecturer Antti Siirtola

Foreword

This thesis has been a pervasive part of my life for the past eight months, and I am very grateful to all who have helped me in laying it to rest. Thanks to my bosses at work, Jyrki Kennilä, Timo Pyhäluoto, and Jesse Pasuri, for arranging my duties so that I can focus on the thesis. Thanks to my colleagues Markku Ahvenjärvi and Veikko Rytivaara for throwing ideas around and kicking off this project.

Thanks to all my peer reviewers for insightful comments, and to Ari Vesanen for the scrutiny. Thank you Antti Siirtola, my thesis supervisor, for providing constant feedback and keeping this study on track. I express my gratitude especially to Satu and Eveliina Anttila for providing me with better working conditions than I could have hoped for, and for keeping me up and running with all the joy they bring into my life.

Tapio Anttila

Oulu, June 3, 2020

Contents

Abstract 2
Foreword 3
Contents 4
1. Introduction 5
2. Research Problem 8
  2.1 Background and objectives for a solution 8
  2.2 Research questions 9
3. Research Methods 11
  3.1 Design Science Research 11
  3.2 The DSR guidelines and process 13
4. Literature Review 17
  4.1 Machine learning 17
  4.2 Log-based anomaly detection 18
    4.2.1 Log parsing phase 19
    4.2.2 Feature extraction phase 20
    4.2.3 Anomaly detection phase 21
5. Existing tools for Log-based Anomaly Detection 23
  5.1 *logparser* 23
  5.2 *loglizer* 23
6. Description of the Dataset 26
  6.1 Pre-processing the initial data 26
  6.2 The final dataset 26
7. First Iteration: Using the *logparser* Toolkit 27
  7.1 Testing LenMa 27
  7.2 Testing SHISO 28
  7.3 Testing AEL 28
  7.4 Parsing the subset 28
8. First Iteration: Using the *loglizer* Toolkit 30
  8.1 Sequence determination 31
  8.2 Training and test data 32
  8.3 A walkthrough of *loglizer* 34
  8.4 Evaluation of *sgologs_alpha* 36
    8.4.1 Using Time sequences and SHISO logs 36
    8.4.2 Using Time sequences and AEL logs 39
    8.4.3 Using PID sequences and SHISO logs 41
    8.4.4 Using PID sequences and AEL logs 42
  8.5 Summary of the first iteration 43
9. Second Iteration 45
  9.1 Changes to *sgologs_alpha* 46
  9.2 Evaluation of *sgologs* 46
10. Discussion 51
  10.1 Answers to the research questions 51
  10.2 Other considerations 54
11. Conclusion 56
  11.1 Limitations 57
  11.2 Future research 57
References 59

1. Introduction

This Master's Thesis is about the development of a log file analysis software tool, with the principal function of detecting anomalies. Taking advantage of existing solutions based on machine learning (e.g. S. He, Zhu, He & Lyu, 2016), the developed tool, dubbed *sgologs*, processes log files from an embedded Linux device, and finds unusual log entries which may indicate abnormal behaviour of the software. Thus, the burden of debugging erroneous behaviour based on log files is mitigated.

A colleague of mine at a midsized ICT company was recently tasked with finding a rare bug in one of our products. Since log files record every single event that is carried out (Landauer, Wurzenberger, Skopik, Settani, & Filzmoser, 2018), with time and state data, they are a natural first stop in the debugging process. However, inspecting the log files is very cumbersome: the behaviour of modern software systems tends to be too complex for a single developer to comprehend, and the sheer volume of logs makes the task extremely time-consuming (S. He et al., 2016). The colleague figured his work would be much more efficient if a software tool would find the anomalies for him, automating the process, resulting in the idea behind this study.

This study utilizes existing, open-source solutions for log-based data mining and anomaly detection. The existing solutions are toolkits, which include several implementations of different methods. The first toolkit is *logparser*, developed by P. He, Zhu, He, Li and Lyu (2016) and Zhu et al. (2019). The second toolkit is *loglizer*, developed by S. He et al. (2016). The existing tools are extensively tested in the context of Linux log files, a topic only briefly noted in Zhu et al. (2019). More specifically, the applicability and performance of unsupervised machine learning methods (discussed more in section 4) with Linux logs are thoroughly investigated in this study.

Other methods than machine learning, e.g. a deterministic algorithm searching for certain keywords or sequences, could be used for the anomaly detection. However, one advantage of machine learning is that it is able to detect previously unknown anomalies (Geijer & Andreasson, 2015). In other words, creating a comprehensive set of keywords for an algorithm to look for would require that all possible anomalies are already known, and new types of anomalies not yet encountered by the creator of the keywords could not be detected. Some argue that focusing only on significant words is not sufficient for a thorough analysis of a system (Landauer et al., 2018). Moreover, systems and their behaviour change over time, changing also the definition of normal behaviour; machine learning can react to such changes, and retain its accuracy (Landauer et al., 2018; Geijer & Andreasson, 2015).

A few of my colleagues and I have discussed the possibilities of the log file analysis software, and we believe that such a tool could be very useful to numerous people here at our company. In addition, this study contributes to research on log-based anomaly detection, but in the specific context of Linux log files.

Log-based anomaly detection endeavours often benefit from unsupervised machine learning for practical reasons (Geijer & Andreasson, 2015). Machine learning algorithms need training material, which can be labelled, i.e. all instances are manually

marked as normal or anomalous (Pietikäinen & Silvén, 2019). However, labelling is usually not a practical possibility (Chandola, Banerjee & Kumar, 2009; S. He et al., 2016). As unsupervised learning does not need labels in the training material, it is an applicable approach in practical settings (S. He et al., 2016). For this reason, unsupervised machine learning is used in this study.

Landauer et al. (2018) state that unsupervised methods can successfully detect anomalies. However, S. He et al. (2016) measured that their anomaly detection tools which leveraged unsupervised learning achieved inferior performance in comparison to tools using supervised learning, i.e. learning based on labelled training material. Zhu et al. (2019) found that log parsing, which refers to the activity of pre-processing the unstructured logs into more structured data, is relatively difficult with Linux logs. This is due to their complex structure and large number of different constant parts in the log entry strings (Zhu et al., 2019). S. He et al. (2016) did not use Linux logs in their study, so the suitability of their toolkit in anomaly detection on Linux systems was left unexplored.

This study aims to discover whether the practical approach of unsupervised learning can be successful in detecting anomalies in Linux logs. The performance of the *loglizer* tools (S. He et al., 2016) with Linux logs is put to the test in order to achieve this. In addition, the performance of the *logparser* tools (P. He et al., 2016; Zhu et al., 2019), and how it affects the anomaly detection, is investigated. Whether the developed tool actually helps in and simplifies the debugging processes of developers is another important consideration.

Since this thesis is an output-based study, where a software solution to a practical problem was developed, it was natural to use Design Science Research (DSR) as the research methodology. The purpose of DSR is designing and creating an artefact, which addresses a real-life organizational problem (Hevner, March, Park & Ram, 2004; Peffers, Tuunanen, Rothenberger & Chatterjee, 2007). Conceptually, the artefact can be anything with a research contribution embedded in the design (Peffers et al., 2007). Here, the artefact is the log file analysis tool, *sgologs*. The tool was developed iteratively, with the evaluation of the artefact and its utility affecting the understanding of the problem, objectives for the artefact, design of the artefact, and so on. Artefact development typically occurs within a build-and-evaluate loop (Hevner et al., 2004). In DSR, the evaluation of the artefact, which can be done based on e.g. functionality, performance, or usability, is as important as its development (Hevner et al., 2004). The performance of the log file analysis tool was simple to measure, by checking how many real anomalies were detected. The final step in a DSR project is the communication of the problem and its relevance, and the designed artefact and its utility, to both management-oriented and technology-oriented audiences (Hevner et al., 2004). The present research process and results are communicated by this thesis.

Overall, Linux log files are challenging for the *logparser* and *loglizer* tools. Still, a parser from the *logparser* toolkit, called AEL, seemed to parse the logs relatively accurately. In addition, unsupervised machine learning models were found to correctly classify anomalies in the Linux log files. However, only two of the four tested models were applicable, and the achieved accuracy scores are rather modest and highly dependent on numerous factors. Nevertheless, if the user is aware of these factors, *sgologs* can direct the user towards abnormal entries in the log file. For example, it is important to be conscious of the fact that the models are best at detecting additional log messages not normally present in the log files. At least to some extent, *sgologs* could operate as a functional troubleshooting tool in real-life settings as well.

The structure of this thesis is as follows. The research problem and questions are set in section 2. Section 3 presents the research method, Design Science Research, and how it is used in this study. Literature review is in section 4, followed by a presentation of the existing tools, *logparser* and *loglizer*, in section 5. Then, the log file dataset is described in section 6. The first iteration of the artefact development is discussed in sections 7 and 8: choosing a log parser and parsing the data is described in section 7, and testing the machine learning models and detecting anomalies with them in section 8. Next, the second iteration is described in section 9. In section 10, the research questions and other considerations are discussed. Finally, section 11 concludes this study.

2. Research Problem

This section describes the research problem in detail. First, the background of the research artefact is discussed in section 2.1, along with the objectives for the tool. Then, research questions are presented in section 2.2.

2.1 Background and objectives for a solution

The log file analysis is based on machine learning. S. He et al., (2016) have presented and evaluated several applicable machine learning approaches for log-based anomaly detection in their paper, and even made an open source release of the studied software tools in the form of the *loglizer* toolkit. It is obviously efficient to take advantage of existing solutions. Initially there were two options for utilizing the *loglizer* tools: they could be used as a kind of blueprint for the design of *sgologs*, or alternatively be used as they are. In the latter case, the developed artefact is a *wrapper*, which prepares the log file data, sends it to a *loglizer* tool, and presents the output. Ultimately it was decided that the log file analysis tool is a wrapper, because getting *logparser* and *loglizer* to work with these Linux log files was well enough work for a thesis project. Input for *sgologs* is a log file, and the tool presents as output the found anomalous sequences in a text file.

The design artefact can be considered successful if it is in fact useful to developers. On a general level this means that the *sgologs* tool considerably streamlines the laborious task of inspecting log files, resulting in more effective analysis workflow for developers. Of course, it is important to remember that even the best tool in this context can only guide a developer towards anomalies in the log files, and the responsibility of how to use the information the tool provides is left for the developer.

The minimum objective for the solution is that anomalies, e.g. erroneous behaviour due to a software bug, are detected with sufficient accuracy and reported. Here, *accuracy* chiefly refers to how many real anomalies are detected. Unsupervised machine learning typically produces a large number of false positives (Landauer et al., 2018). This indicates that the machine learning reacts to instances which are not anomalous, resulting in unnecessary data in the output, but also that most, if not all, real anomalies are detected. In this study it is assumed that false positives are better than undetected anomalies, also called false negatives; if the machine learning is too strict about false positives, and thus too careful with the classification, real anomalies may not be detected (Landauer et al., 2018).

Before the logs can be fed into the anomaly detection, they require pre-processing, i.e. log parsing (S. He et al., 2016). As mentioned, tools in the *logparser* toolkit have some difficulties with Linux logs, the best accuracy being 0.701, whereas other logs could be parsed with accuracies higher than 0.9 (Zhu et al., 2019). The measurements are of course highly dependent on the used log files, and the Linux logs used in this study could produce different results. This topic is investigated in this study, but log parsing is not measured as in Zhu et al. (2019). The accuracy scores are based on comparison to the “ground truth”, i.e. a manually parsed log file (Zhu et al., 2019). However,

constructing such a log file to present the ground truth is not feasible in a one-person thesis project. Moreover, it is unclear whether a log parsing accuracy of e.g. 0.701 considerably hinders the anomaly detection. The effect of different parsing accuracies on the anomaly detection is investigated in this study, as mentioned in section 7.4.

It is also desirable for the tool to be easy to use, fast, and reliable. If *sgologs* processes for a long time, or presents false results, it will not be useful to developers. Found anomalies, or the absence of anomalies, are the main message of the tool's output. *Sgologs* informs in the command prompt or terminal if anomalies are found, and mentions were the output files are stored.

2.2 Research questions

Machine learning algorithms need training material, i.e. log files, to learn the expected and anomalous patterns in the logs. The patterns can be already identified in the training data, but it is not an absolute requirement (S. He et al., 2016). This topic is discussed more in section 4. Labelling the training data is rarely a practical possibility (S. He et al., 2016). For example, for the embedded device acting as the source of the log files, each boot produces relatively unique logs, regardless whether anomalous behaviour occurs or not. Thus, the classification of expected and anomalous patterns would be very time-consuming and difficult, when a baseline for comparisons is lacking. Therefore, it is reasonable to rely on unlabelled training data. However, the functionality of the anomaly detection needs to be verified by manually checking if the found anomalies are correct and if all real anomalies are found in a limited selection of log files.

The fact that the *loglizer* tools group log messages into log sequences creates a challenge. Developers typically inspect the log files as they are, without converting them into specific sequences of any kind. Consequently, it would be difficult for them to identify which sequence is normal or not. The compromise made in this situation is to check how many sequences that contain log messages with the word *error* is marked as anomalous. Focusing on errors greatly simplifies the validation of the anomaly detection results, as data and knowledge about possible labels is non-existent. This approach overlooks the fact that some anomalies may not be errors per se; however, errors are exactly what developers typically search for in the log files. Whether the amount of detected sequences that contain errors is a valid representation of the anomaly detection accuracy is unclear. The first research question is

> *RQ1: Can the accuracy of the anomaly detection be effectively verified by the amount of detected errors?*

S. He et al. (2016) measured that unsupervised machine learning models achieve inferior performance compared to supervised models. It is relevant to consider whether unsupervised learning is good enough for this purpose, or would it be better to take the trouble of generating label files to enable the use of supervised learning. Fortunately, unsupervised learning supposedly has shown promise in anomaly detection problems (S. He et al., 2016; Landauer et al., 2018). Anomaly detection techniques tend to suffer from high amounts of false positives (Landauer et al., 2018), which are likely to obscure real anomalies from the output. As discussed above, false positives should not be a problem, as long as real anomalies are detected as well. Naturally, the desired accuracy for the anomaly detection is that all real anomalies are detected, and the smaller the amount of false positives is the better. The second research question is

RQ2: How accurate is the anomaly detection with unsupervised machine learning and Linux log files?

3. Research Methods

This section introduces the Design Science Research (DSR) methodology and its usage in this study. A general picture of DSR is presented in subsection 3.1. Then, the DSR guidelines (Hevner et al., 2004), their application in this study, and how they relate to the DSR process activities (Peffers et al., 2007), are presented in 3.2.

3.1 Design Science Research

Instead of reality-describing research in the style of social and natural sciences, DSR is about creating an artefact for human purposes, addressing a real-life problem (Peffers et al., 2007). DSR is often compared to behavioural science, where an IT artefact is often the *object of study*, whereas design science creates and evaluates IT artefacts (Hevner et al., 2004). "Such artifacts are represented in a structured form that may vary from software, formal logic, and rigorous mathematics to informal natural language descriptions" (Hevner et al., 2004, 77). The design artefact, and the proof of its usefulness, is central in DSR (Peffers et al., 2007). In this study, the designed artefact is the log file analysis software tool.

By the common DSR artefact classification, the log file analysis tool is an instantiation, instead of a construct, a model, or a method. *Constructs* essentially are the symbols used to define problems and solutions. For example, mathematics relies on the constructs of Arabic numbers and zero. *Models*, i.e. representations of the problem domain, can be created with the constructs. More than a model, but not quite an instantiation, a *method* could be e.g. an algorithm. Finally, an *instantiation* is a functional implementation, or a prototype, of the solution. (Hevner et al., 2004.)

In an organization, the strategies for business and information technology, and the infrastructure for organizational processes and information systems, are largely aligned. Design is required for the effective transition from strategy to infrastructure in both organizational context and information systems context, as illustrated in Figure 1. (Hevner et al., 2004.)

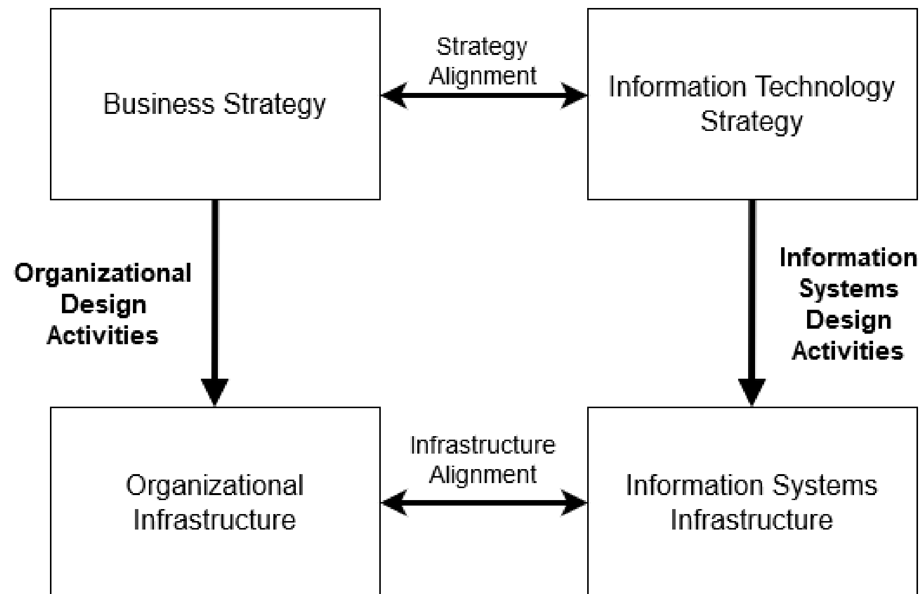

Figure 1. Organizational Design and Information Systems Design Activities. Adapted from Hevner et al. (2004, Figure 1, p. 79).

In addition to considering and leveraging existing research in DSR, and information systems research in general, it is important to remember the environment where the research is happening, also known as problem space. Business needs, i.e. goals and problems within an organization, are the factors which fuel the IS research endeavour. The research thus has *relevance* to the environment. Likewise, the base of existing knowledge provides the researcher with the scientific foundations and methods for conducting the study. In other words, the knowledge base helps in ensuring the research is performed *rigorously.* The new research then gives back to the environment, by applying the solution in the organization, and to the knowledge base, by presenting the results of the study to the research community. These relationships are illustrated in Figure 2. (Hevner et al., 2004.)

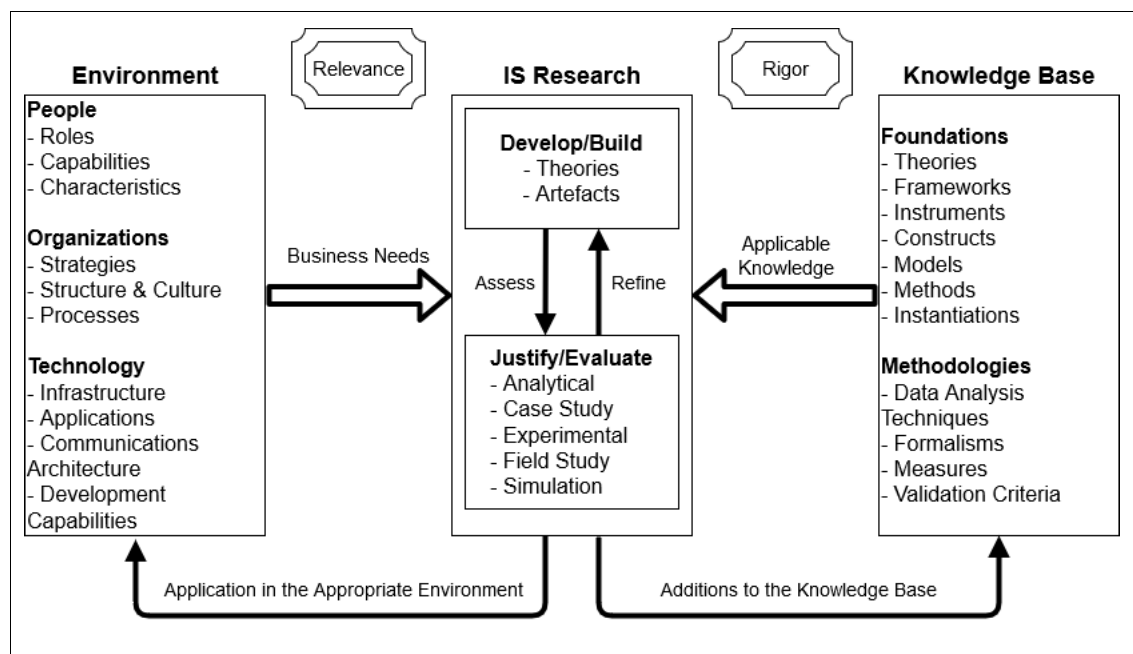

Figure 2. Information Systems Research Framework. Adapted from Hevner et al. (2004, Figure 2, p. 80).

In DSR, the evaluation of the design artefact is as important as building the artefact. Evaluation produces feedback information on the artefact, and a better understanding of the problem, thus making it possible to improve not only the quality of the artefact but also the quality of the design process. This loop of building and evaluating, also depicted at the centre of Figure 2, is typically iterated through several times. (Hevner et al., 2004.)

What differentiates Design Science Research from routine design is the fact that DSR addresses unsolved problems in innovative ways, or solved problems in more effective or efficient ways. Routine design, on the other hand, is simply the application of existing knowledge to known problems, without any contribution to the knowledge base of existing research. (Hevner et al., 2004.)

3.2 The DSR guidelines and process

Hevner et al. (2004) presented seven guidelines for DSR, which describe the requirements for effective Design Science Research. The guidelines are summarized in Table 1. Following Peffers et al. (2007), the design science process has six steps: problem identification and motivation, definition of the objectives for a solution, design and development, demonstration, evaluation and communication. The order of these activities is not strict, and the process and the activities can be repeated in an iterative fashion (Peffers et al., 2007). The process activities are summarized in Table 2. This process model is naturally followed in this study. Next, the guidelines (and related process activities) are described in detail, followed by information about their application in this study.

Table 1. Design Science Research Guidelines (Hevner et al., 2004, p. 88)

Guideline	**Description**
Guideline 1: Design as an Artefact	Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
Guideline 2: Problem Relevance	The objective of design-science research is to develop technology-based solutions to important and relevant business problems.
Guideline 3: Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
Guideline 4: Research Contributions	Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.
Guideline 5: Research Rigor	Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.
Guideline 6: Design as a Search Process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.
Guideline 7: Communication of Research	Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

Table 2. The Design Science Research Process (Peffers et al., 2007)

Activity	Description
Activity 1: Problem identification and motivation	Define the research problem and justify the value of the solution.
Activity 2: Definition of objectives for a solution	Define the objectives for a solution based on the problem definition, and knowledge of what is possible and feasible.
Activity 3: Design and development	Determine the functionalities of and create an artefact, where a research contribution is embedded in the design.
Activity 4: Demonstration	Use the artefact e.g. in experimentation or simulation to solve one or more instances of the problem.
Activity 5: Evaluation	Observe and measure how well the artefact solves the problem.
Activity 6: Communication	Present the problem and its importance, and the artefact and its design and utility.

Guideline 1: Design as an artefact. The creation of an artefact which addresses an organizational problem is the purpose of DSR. Artefacts are rarely full-grown information systems (Hevner et al., 2004). If the artefact is an instantiation, it should prove the feasibility of both the design process and the designed product (Hevner et al., 2004). Conceptually the artefact can be anything, as long as a research contribution is embedded in the design (Peffers et al., 2007). It is also relevant to consider the objectives of the artefact, aligning this guideline with Activity 2. The objectives can be quantitative or qualitative, e.g. a determination of how a new solution is better than current ones, or a description of how a new artefact is expected to support solutions to problems (Peffers et al., 2007).

In this study, the design artefact is the log file analysis tool. As an instantiation, it proves the feasibility of the process and the product: the artefact mitigates the burden of log file analysis, thus showing that the burden of log file analysis can in fact be mitigated. The objectives for the artefact are described in section 2.

Guideline 2: Problem relevance. DSR must address a known problem within an organization, namely a difference between a goal state and the current state (Hevner et al., 2004). The first activity in the DSR process model (Peffers et al., 2007) is to define the research problem and to justify the value of the solution. It may be necessary to conceptually atomize the problem, so that the solution may capture its complexity. Justifying the value of the solution motivates the researcher and the audience of the research and clarifies the researcher’s understanding of the problem (Peffers et al., 2007).

The log file analysis tool developed in this study addresses the problem of a considerably laborious task. The artefact streamlines this task, leaving more time for more important tasks, and making the work of developers who use the tool more efficient. The laborious task faces developers quite often, underlining the relevance of the problem.

Guideline 3: Design evaluation. As mentioned above, the evaluation of the artefact is as important as the development of the artefact. Evaluation can be done based on e.g. functionality, performance, or usability. The artefact should also be integrated into the business environment. (Hevner et al., 2004.)

Observing and measuring how well the artefact solves the problem can be done during and after the demonstration phase. Generally evaluation (Activity 5) is about comparing the objectives to the actual observed results. As an example evaluation can take the form of quantitative performance measures or results of satisfaction surveys. After the evaluation it is possible to iterate back to Activity 3 (design and development) and attempt to improve the artefact. (Peffers et al., 2007).

The log file analysis tool is evaluated by observing its performance with the log parsing and anomaly detection (both discussed in more detail in section 4). In addition, the usability of the tool, i.e. how much it helps developers in their log inspection, is discussed with several developer colleagues. I will personally advertise the tool within the company, hopefully raising interest at the management level as well.

Guideline 4: Research contributions. Effective DSR must provide a contribution, be it the artefact itself, or e.g. the used design methodology (Hevner et al., 2004). The artefact itself is the major contribution of this study. In addition, this study provides information about how the S. He et al. (2016) tools perform with Linux logs.

Guideline 5: Research rigor. The construction and evaluation of the artefact must be done with scientific rigor, meaning e.g. transparent and reproducible methods (Hevner et al., 2004). The design artefact of this study is based on the toolkit from S. He et al. (2016), a product of extensive research. Similarly, the construction and evaluation methods used by S. He et al. (2016) also aid the construction and especially the evaluation of the present artefact.

Guideline 6: Design as a search process. The sixth guideline refers to the iterative process of DSR: evaluating the solution to the problem refines the problem itself, which again refines the requirements for the solution, and so on (Hevner et al., 2004). Adhering to the process activities, after the artefact has been demonstrated in Activity 4 and evaluated in Activity 5, it is possible to iterate back to design and development (Activity 3) or even to problem identification (Activity 1) or objective definition (Activity 2) (Peffers et al., 2007). The development of the log file analysis tool was iterative. For example, the first version of the tool, *sgologs_alpha*, needed parsed, structured log files as input, and could only use session windows in determining log event sequences. The second iteration began by going back to Activity 3 and implementing new features.

Guideline 7: Communication of research. The research and its results must be presented to both technology-oriented and management-oriented audiences. Technology-oriented audiences need this information in order to be able to use the artefact in the appropriate context and to enjoy its benefits. Management-oriented audiences wish to know whether the organization should be committed to constructing and using the artefact, so the importance of the problem and the novelty of the solution should be emphasised in the communication. (Hevner et al., 2004.)

Communication (Activity 6) includes the presentation of the problem and its importance as well as the artefact and its design and utility. Communication typically takes the form of a research paper, or alternatively a thesis. (Peffers et al., 2007.)

The present research and results are communicated with this thesis. The thesis provides the relevant information to both technology- and management-oriented audiences.

Figure 3 presents the DSR process model in visual form. At the bottom, the possible research entry points, i.e. possible factors which initiate the process, are also visible. The research entry point of this study is highlighted, and the relevant specific tasks of the activities are written out. The texts between the activities represent the required resources for the succeeding (and preceding) activity.

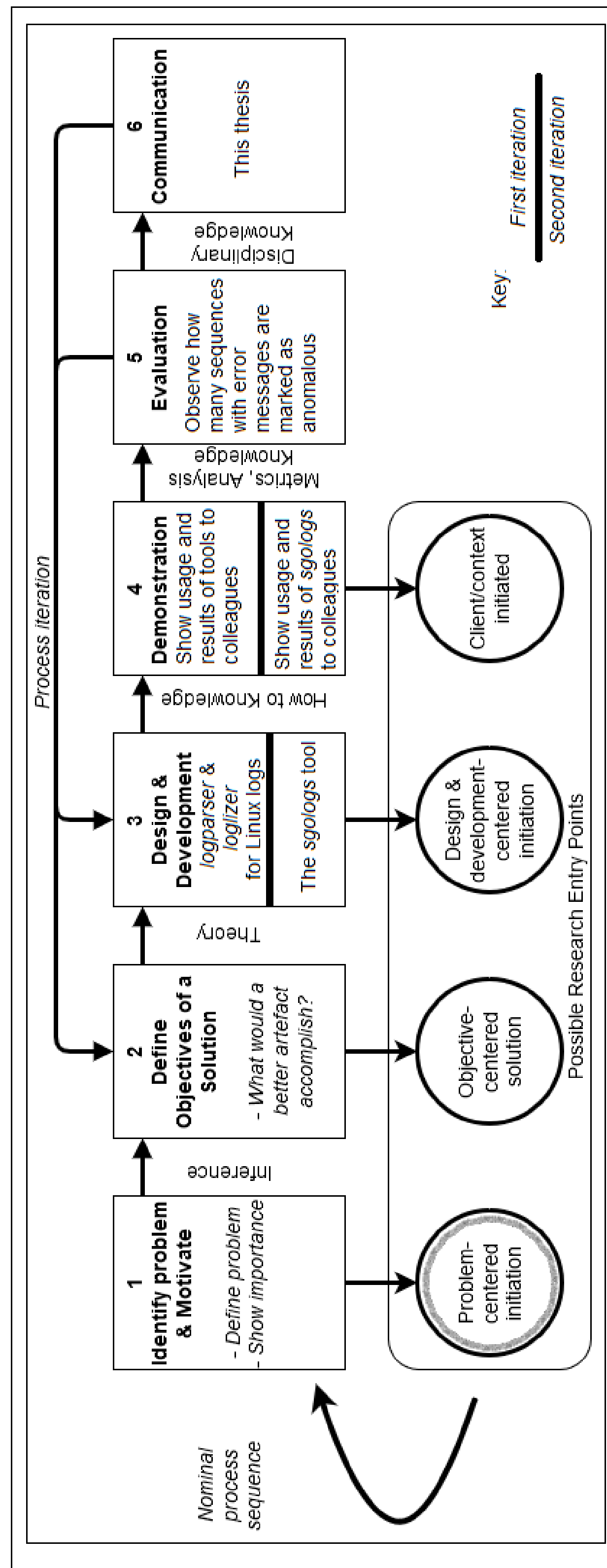

Figure 3. Design Science Research Process Model. Adapted from Peffers et al. (2007, Figure 1, p. 54).

4. Literature Review

This section presents earlier studies on topics relevant to this research. A brief overview on machine learning is presented in 4.1, followed by more extensive discussion on the specific topic of log-based anomaly detection in section 4.2.

4.1 Machine learning

Machine learning (ML) refers to the ability for computer programs to analyse data, extract information, and learn from it. How machine learning algorithms “learn” is essentially about creating and recognizing patterns in training data. For example, in the context of this study, the training data is actual log files, which the algorithms process to generate models for anomaly detection. The intelligence of machine learning is based more on modelled data, instead of the learning algorithms themselves (Pietikäinen & Silvén, 2019). Machine learning is successful with problems which are difficult to solve with purely algorithmic approaches, such as speech and image recognition (Pietikäinen & Silvén, 2019). Different ML approaches have been leveraged in countless domains, for example in software fault prediction (Malhotra, 2015), solar radiation forecasting (Voyant, Notton, Kalogirou, Nivet, Paoli, Motte & Fouilloy, 2017), text-document classification (Khan, Baharudin, Lee & Khan, 2010), network intrusion detection (Tsai, Hsu, Lin & Lin, 2009) and financial market prediction (Henrique, Sobreiro & Kimura, 2019).

Machine learning methods are commonly divided into three categories: supervised learning, unsupervised learning, and reinforcement learning (Pietikäinen & Silvén, 2019). Supervised learning is based on examples: the training material input is labelled, i.e. categories in the training data are already recognized and identified (Pietikäinen & Silvén, 2019). The requirement of labelled training material presents some problems. Labelling the data may not be feasible, since e.g. all the categories in the data should be known beforehand (S. He et al., 2016; Geijer & Andreasson, 2015), and accuracy of the algorithms suffers from incorrectly labeled training material (Pietikäinen & Silvén, 2019). Labelling is usually done by a human expert, making accurate and representative labelling prohibitively expensive (Chandola et al., 2009). However, strong training material can result in very accurate results (S. He et al., 2016). Popular supervised methods include logistic regression and decision trees (S. He et al., 2016; Pietikäinen & Silvén, 2019).

On the contrary, unsupervised learning does not require labels in the training data. The underlying principle is creating a depiction of the structures in the data, where similar inputs are located near each other (Pietikäinen & Silvén, 2019). Unsupervised methods are usually more feasible in practical settings, but unfortunately tend to achieve lower performance than supervised methods (S. He et al., 2016). Although, Landauer et al. (2018) ambiguously argue that unsupervised methods are able to detect anomalies on unlabelled data. It is important to note that verifying the accuracy of unsupervised methods is difficult if the categories of the data are completely unknown (Landauer et al., 2018). Popular unsupervised machine learning methods include autoencoders and Principal Component Analysis (PCA) (S. He et al., 2016; Pietikäinen & Silvén, 2019).

Semi-supervised learning is a combination of supervised and unsupervised learning. Training data typically has labels for normal behaviour (Chandola et al., 2009). Some anomalous categories are first determined with unsupervised learning, and vague categories are verified by human input (Pietikäinen & Silvén, 2019). As human input is required, semi-supervised learning may face similar challenges as supervised learning, albeit it reduces the amount of human labour (Pietikäinen & Silvén, 2019).

In reinforcement learning, a software agent takes actions in an environment, and receives positive or negative feedback (Pietikäinen & Silvén, 2019). The goal of the machine learning is to determine the solution which receives the most positive feedback (Pietikäinen & Silvén, 2019).

A popular subset of machine learning, deep learning, has been very successful in problems such as speech or image recognition. The method extracts features from the raw data with multiple layers; e.g. in image recognition, lower layers capture features such as edges, and higher layers identify more specific features, such as letters or faces. One major advantage of deep learning over traditional machine learning is that these layers of features need not be engineered by humans, as they are learned from the data. In other words, deep learning is able to process natural, *unstructured* data. (LeCun, Bengio & Hinton, 2015.)

4.2 Log-based anomaly detection

As Chandola et al. (2009) define it, anomaly detection is about finding patterns in data that do not correspond to expected behaviour. There is an abundance of possible applications, such as fraud detection for credit cards, insurance, or health care, intrusion detection for cyber-security, fault detection in safety critical systems, and military surveillance for enemy activities (Chandola et al., 2009). All anomaly detection approaches are essentially about defining a representation of normal behaviour, and declaring anything that does not belong into this representation as an anomaly. However, in practice, anomaly detection is not as straightforward (Chandola et al., 2009). Even defining the normal behaviour can be challenging: often the boundary between normal and anomalous behaviour is not precise, and normal behaviour may evolve over time, requiring updates in the data models (Chandola et al., 2009). The latter consideration is also true for anomalous behaviour. Moreover, malicious actions often mask themselves as normal (Chandola et al., 2009), for example in cyber-attacks. The availability of labelled data for the training and the validation of the models is another challenge (Chandola et al., 2009; S. He et al., 2016).

Anomalies appear in data for a wide variety of reasons. Suspicious activity in network traffic may indicate that a cyber-attacker has intruded the network (Dunaev & Zaytsev, 2019). Malignant tumours result in anomalous MRI images (Chandola et al., 2009). Possible causes for software execution anomalies are hardware problems, network communication congestion or software bugs (Fu, Lou, Wang & Li, 2009). There are several possible approaches for anomaly detection, such as detection at the operating system level, without having to rely on e.g. event logs (Bovenzi, Brancati, Russo, & Bondavalli, 2015). Nevertheless, log-based anomaly detection has become a common approach and attracted a lot of academic interest. With software, the advantage of log files is that they record every event that is carried out (Landauer et al., 2018; Leal-Aulenbacher & Andrews, 2013), being the only source of information about a program's execution and behaviour with time and state data (Dunaev & Zaytsev, 2019).

Therefore, the logs are the information that can be processed to detect anomalous situations (Dunaev & Zaytsev, 2019; Cheng & Wang, 2014).

However, the manual inspection of log files is often unfeasible due to numerous factors. Software systems and their behaviour tend to be too complex for a single developer to comprehend (Fu et al., 2009; S. He et al., 2016; Leal-Aulenbacher & Andrews, 2013). Thus, analyzing log events related to a component one is not familiar with are of little value. Moreover, the sheer volume of logs from contemporary software systems exceeds what a single developer can effectively handle (Fu et al., 2009; S. He et al., 2016; Geijer & Andreasson, 2015). In some cases, the logs are produced at rates measured in terabytes or petabytes (Mavridis & Karatza, 2017). Finally, software components can have drastically different fault tolerance mechanisms; for example, failed tasks may be rerun, or speculative tasks may be killed to improve performance (S. He et al., 2016). In such cases, finding suspicious log messages can be very difficult. Log file analysis tools should work autonomously, investigating the contents of the log files instead of any labels they have received: tools that require human input also require that e.g. all possible anomalies are already known (Geijer & Andreasson, 2015). There is definitely demand for log analysis methods for anomaly detection (S. He et al., 2016).

As a side note, log file analysis tools that operate in cloud computing settings also exist. The popular Apache Hadoop and Spark are good examples of such tools. However, they are not solely focused on anomaly detection. (Mavridis & Karatza, 2017.)

S. He et al. (2016) present an overall framework for log-based anomaly detection. The process consists of four phases: log collection, log parsing, feature extraction, and anomaly detection. Most log-based anomaly detection studies, such as Lou, Fu, Yang, Xu and Li (2010), follow this process, possibly using different names for the phases. Log parsing, feature extraction, and anomaly detection are each discussed in detail in the following subsections.

4.2.1 Log parsing phase

Logs cannot be fed into machine learning or data mining models as they are. This is because logs are unstructured, so a crucial first step is to parse log messages into structured data (Zhu et al., 2019). As mentioned above, logs generated by modern software systems tend to be very large, making manual parsing unfeasible, even with the help of e.g. regular expressions. Regular expressions have to be created manually anyway, and also constantly updated as the system evolves (P. He et al., 2016). Fortunately, numerous open-source automated log parsing solutions, which learn from the system and evolve with it, exist. P. He et al. (2016) and Zhu et al. (2019) have released an open-source toolkit, including numerous techniques for log parsing, called *logparser*. *Logparser* will be presented in detail in section 5.

Fu et al. (2009) refer to their log parsing step as log key extraction, which is essentially the same activity. They justify the usage of log keys with two aspects: typically each log key type corresponds to one log print statement in the source code, resulting in log key sequences representing the execution path, and the number of log keys is finite, whereas theoretically there is no limit to the number of unique log messages (Fu et al., 2009). P. He et al. (2016) and Zhu et al. (2019) included Fu et al.'s (2009) method in their *logparser* toolkit.

Log messages typically include fields such as timestamp, verbosity or severity level (e.g. INFO or ERROR), and raw message content, recording what has happened during system operation (P. He et al., 2016). The raw message content can be further divided into constant part and variable part. For example, in the log message "*Received block blk_-56272528 of size 671064 from /10.251.91.84*" the constant part is "*Received block <*> of size <*> from /<*>*" and the variable part is the parameters *blk_-56272528*, *671064* and *10.251.91.84* (Zhu et al., 2019). The constant part could also be referred to as *log key*, as in Fu et al. (2009). Making extractions such as this is key in introducing structure into the logs. Occasionally, this activity is referred to as frequent pattern mining, a common data mining technique, as in Cheng and Wang (2014). Studies often focus solely on the log keys to detect anomalies, while some approaches also take timestamp and parameter values into consideration (Du, Li, Zheng & Srikumar, 2017).

The extracted constant parts of log messages are clustered into a list of log event templates, e.g. "*Event 3: Received block <*> of size <*> from /<*>*". The structured log will then contain a sequence of these events with their occurring times, e.g. "*2020-01-16 13:18:59 Event 3*". Now, when the log is structured, it can be easily processed, for example by machine learning-based anomaly detection methods. It is important to remember that log mining (e.g. log-based anomaly detection) is effective only when the log parsing accuracy is high enough. (P. He et al., 2016.)

Unfortunately, as found in Zhu et al. (2019), Linux logs are difficult to parse accurately, due to their complex structure and the large amount of event templates. The LenMa tool, based on Shima (2015), provided the best accuracy of 0,701 (Zhu et al., 2019).

4.2.2 Feature extraction phase

Fu et al. (2009) use Finite State Automation (FSA) to model the execution path of the system, and construct models representing normal system behaviour. This way, the anomalies can be detected by comparing new log sequences to the FSA models (Fu et al., 2009).

S. He et al. (2016) separated log data into various groups, each group representing a log sequence, using different grouping techniques. Then, feature vectors (or event count vectors) were created for each sequence, and the vectors were used to form a feature (event count) matrix. The matrix is required as input for the anomaly detection models.

The grouping techniques used by S. He et al. (2016) were fixed windows, sliding windows, and session windows. Log events that occurred in the same window were regarded as a log sequence. Both fixed windows and sliding windows are based on the timestamp of each log event. Fixed windows only have one attribute, size (e.g. one hour), whereas sliding windows also have the step size attribute (e.g. five minutes). With these parameters, as an example, two hours has space (or rather time) for two fixed windows, and 13 sliding windows. Log events are likely to duplicate in multiple sliding windows due to overlap, as the step size is in general smaller than the window size (S. He et al., 2016). Instead of timestamps, session windows use identifiers to mark execution paths, with a unique identifier to each session window. S. He et al. (2016) found that sliding windows is the most accurate grouping technique.

Kim, Minsik, Kim, Cho, and Kang (2019) used user behaviour modelling for insider threat detection. For example, they used user activity logs to extract candidate features, such as the number of USB connections per day (Kim et al., 2019). In addition to user

activity logs, Kim et al. (2019) also used email topic modeling to create an email content dataset, and information about how users send/receive information to create an email communication network dataset.

4.2.3 Anomaly detection phase

The bases of three supervised methods are presented in S. He et al. (2016): logistic regression, a statistical model, decision tree, a tree structure diagram, and Support Vector Machine (SVM), a supervised learning method for classification. Additionally, S. He et al. (2016) present three unsupervised anomaly detection methods: log clustering, Principal Component Analysis (PCA) and invariants mining. S. He et al. (2016) also evaluate the efficiency of all these methods.

Supervised methods achieve high precision (the percentage of how many reported anomalies are correct), while recall (the percentage of how many real anomalies are detected) is influenced by the analysed datasets and window settings. It should be noted that *accuracy* is often used as an umbrella term, encompassing both precision and recall: 100% accuracy requires and indicates that precision and recall are also 100%. Overall, SVM seems to be the most accurate supervised anomaly detection method. Unfortunately, even though unsupervised methods tend to be more applicable in practical settings, they generally achieve inferior performance compared to supervised methods. However, invariants mining appears as a promising method with stable performance, and window settings do not seem to affect the results as they did with supervised methods. (S. He et al., 2016.)

Unsupervised anomaly detection methods, except for PCA, were measured to be much more time-consuming than supervised methods. Methods in both categories, except the unsupervised log clustering, scale linearly as log size increases. However, like log clustering, invariants mining requires optimizations to be able to handle large datasets. (S. He et al., 2016.)

Cheng and Wang (2014) also used PCA in their study on communication network anomalies. They achieved impressive accuracies, the minimum being 66% and the average near 80%. Moreover, the rate of false positives was admirably low: the maximum value was only 0.49%.

Kim et al. (2019) fed their candidate feature sets from their three datasets (user activity-, email content-, and email communication network dataset) into one-class classification algorithms. Since the amount of abnormal cases of user activity is typically very small, it is practical to use one-class classification, which only uses the normal class data to learn their common characteristics (Kim et al., 2019). Kim et al.'s (2019) trained model then predicts the likelihood of a newly given instance being a normal class instance. The algorithms used were Gaussian density estimation, Parzen window density estimation, PCA, and K-means clustering. Kim et al.'s (2019) framework performed reasonably well: for example, when considering 30% of the most suspicious instances, more than 90% of abnormal behaviours in the user activity dataset, and 65.64% in the email content dataset, were detected. However, their anomaly detection models were trained independently based on each dataset, and Kim et al. (2019) argue that better integration of the results, and utilizing the knowledge of experts, would possibly achieve a better performance.

Landauer et al. (2018) used their own incremental clustering algorithm for intrusion detection. Incremental cluster methods dynamically allocate incoming data points to existing clusters, or, if the distance to the nearest cluster exceeds a certain threshold, declare them as outliers (Landauer et al., 2018). In addition to clustering, Landauer et al. (2018) detected anomalous behaviour with cluster evolution and time-series analysis. When applied on the evolutions of individual clusters, their anomaly detection showed promising performances. However, their method only detects dynamic changes, overlooking anomalies that occur within a single time window, and "the problem of rather *high amounts of false positives that all anomaly detection techniques suffer from* [emphasis added] remains unsolved" (Landauer et al., 2018, 115).

Du et al. (2017) use deep learning, more specifically a type of recurrent neural networks (RNN) called long short-term memory (LSTM), for log-based anomaly detection. The DeepLog model is able to detect execution path anomalies, by inspecting the log key sequences, but also parameter value and performance anomalies. This is possible because the parameter and timestamp values extracted from the log entries are taken into account in the anomaly detection. DeepLog also models the execution path of the system, like Fu et al. (2009) did with FSA. This modelling is inspired by invariants mining (Lou et al., 2010), and is performed with density-based clustering as well. Overall, DeepLog achieves a better performance than previous methods, such as PCA, which are also unable to detect parameter value and performance anomalies. (Du et al., 2017.)

5. Existing tools for Log-based Anomaly Detection

This section presents an overview of the *logparser* and *loglizer* toolkits. *Logparser*, developed by P. He et al. (2016) and Zhu et al. (2019), is described in section 5.1, and *loglizer*, developed by S. He et al. (2016), is described in section 5.2.

5.1 *logparser*

Logparser[1] is an open-source toolkit, used to convert raw log messages into a sequence of structured events (Zhu et al., 2019). The toolkit automates the process of extracting event templates, and provides a decent selection of different log parsing methods (P. He et al., 2016; Zhu et al., 2019).

Logparser contains 13 log parsing methods proposed by researchers (such as Fu et al., 2009) and practitioners, five of which are based on open-source tools. The input/output interface is unified for all the different methods, which are also wrapped into a single Python package. For all the tools, input is a raw log file, and the output is (1) a structured log file and (2) an event template file with aggregated event counts. (Zhu et al., 2019.)

As mentioned above, Linux log files are challenging for automated log parsing (Zhu et al., 2019). The highest parsing accuracy in Zhu et al.'s (2019) measurements was achieved with the LenMa tool, which is based on Shima (2015). The rounded accuracy of LenMa was measured to be 0.701 (Zhu et al., 2019). The second and third best tools were SHISO (Mizutani, 2013) and AEL (Jiang, Hassan, Flora & Hamann, 2008), which achieved rounded accuracies of 0.701 and 0.673, respectively (Zhu et al., 2019).

5.2 *loglizer*

Loglizer[2] is a log analysis toolkit for automated anomaly detection, based on machine learning. At the time of its release, the toolkit included six tools: three with supervised machine learning methods, and three with unsupervised machine learning methods. However, three additional tools using unsupervised models have been released recently: LOF (Local Outlier Factor), One-class SVM, and Isolation Forest. Moreover, two more tools, DeepLog (based on deep learning) and AutoEncoder, are currently in development. (Logpai, 2020.)

It is relevant to highlight here that the anomaly detectors by S. He et al. (2016) work with log sequences, instead of individual log messages. This means that the machine learning algorithms classify collections of log entries, and thus never mark a single log

---

[1] https://github.com/logpai/logparser

[2] https://github.com/logpai/loglizer

message as anomalous. As described in section 4.2.2, S. He et al. (2016) used different windowing techniques for generating the log sequences. The specifics of the windowing techniques used in this study are presented in section 8.1.

As discussed in section 4, unsupervised machine learning is more applicable in this context, and therefore the unsupervised tools were focused on. The usage of four unsupervised tools is conveniently demonstrated within the *loglizer* toolkit, with implementations which do not assume that label data is available. These four tools (InvariantsMiner, IsolationForest, LogCluster, and PCA) were subject to testing in this study, instead of using only the three from S. He et al. (2016). Next, the underlying anomaly detection techniques of the tools are described in more detail.

LogCluster tool is based on work by Lin, Zhang, Lou, Zhang and Chen (2016), originally developed to identify online system problems (S. He et al., 2016). Clustering is a relatively common, primarily unsupervised technique, where similar data instances are assigned into clusters (Chandola et al., 2009). This technique relies on the assumption that normal instances belong to a definable cluster, and lie close to their closest cluster centroid, whereas anomalous instances do not (Chandola et al., 2009). However, clustering algorithms are usually optimized to find clusters, not anomalies, and tend to have trouble with clusters of anomalies (Chandola et al., 2009). LogCluster is trained in two phases: knowledge base initialization phase, where normal and abnormal clusters are generated, and online learning phase, where the clusters are further adjusted (S. He et al., 2016). In the anomaly detection, the distance of a new log sequence to its nearest cluster is computed. If the smallest distance is larger than some threshold, or if the nearest cluster is an abnormal cluster, the log sequence is reported as an anomaly (S. He et al., 2016).

Principal Component Analysis (PCA) is a spectral anomaly detection technique, which tries “to find an approximation of the data using a combination of attributes that capture the bulk of the variability in the data” (Chandola et al., 2009, 37). In other words, PCA is about projecting high-dimension data to a new coordinate system composed of k principal components (k being less than the original dimension), preserving the major characteristics of the original data (S. He et al., 2016). Log sequences are vectorised as event count vectors, and PCA is used to find patterns between the dimensions of the vectors (S. He et al., 2016). Then, a projection of an event count vector is calculated, and if the length of the projection is larger than some threshold, the vector is classified anomalous (S. He et al., 2016). The PCA tool in the *loglizer* toolkit is based on Xu, Huang, Fox, Patterson and Jordan (2009), where PCA was used precisely in log-based anomaly detection (S. He et al., 2016).

Invariants Mining focuses on program invariants, linear relationships that always hold during the runtime of a system. For example, files need to be closed after they were opened, so log entries with phrases “open file” and “close file” appear in pairs and represent a lineal relationship. If the number of “open” and “close” log events in an instance is not equal, the linear relationship is violated, and the instance is marked anomalous. The *loglizer* tool is based on Lou et al. (2010), and as the name suggests, the invariants (i.e. the linear relationships) are extracted from the log files. Each new log sequence is reported as an anomaly if it disobeys at least one invariant. (S. He et al., 2016.)

Typically, anomaly detection methods construct a profile of normal instances, and instances which do not conform to this profile are identified as anomalies, but Isolation Forest has a different approach. Taking advantage of the fact that anomalies are few and

different compared to normal instances, Isolation Forest isolates anomalies instead of profiling normal instances. No distance or density calculations are required, reducing computational cost. Moreover, Isolation Forest has a linear time complexity, and can be scaled up to large and/or high-dimensional data sets. (Liu, Ting & Zhou, 2008.)

6. Description of the Dataset

This section describes the log dataset used in this study. The initial data was pre-processed (or pre-pre-processed) to prepare it for the log parsing, as explained in section 6.1. The resulting final log file dataset is summarized in section 6.2.

6.1 Pre-processing the initial data

The aforementioned colleague had collected the logs from the Linux journal, which records information e.g. about all kernel and userspace processes (see e.g. "systemd," 2020). The logs were collected from 9 devices, running two different builds of custom embedded Linux. Here, the two builds are dubbed "A" and "B". Each device stored its journal logs from numerous consecutive reboots into a single file, resulting in files up to 3.7 GB in size. The files were stored in two directories, A and B, depending on which build of the software they were collected from. In the files, a few rows of free-form debug prints, written by the colleague, preceded the actual log entries of each reboot cycle.

First, the debug prints were removed, leaving only the entries that conformed to Linux log format. Second, the massive files of consecutive logs were split into multiple files, with each resulting file containing the log entries of a single reboot. Both of these tasks were straightforward, and carried out by simple Python scripts.

6.2 The final dataset

After the initial processing, the complete dataset contained 275 605 log files, taking up 34.9 Gigabytes of storage space. The amount of rows, i.e. log entries, in each log file varied from 1500 to 2500, occasionally being as high as 35 000. Typically, a single log file contained about 1750 log entries.

Needless to say, the amount of data is definitely sufficient for the purposes of this study. Processing all the log files was obviously not necessary or feasible, and therefore a randomly selected subset of the log files was used. The subset contained 9040 log files from both directories (A/B). The subset was used in further processing and analysis, first in testing the *logparser* tools. The complete dataset and its subset are summarized in Table 3.

Table 3. Summary of the dataset.

COMPLETE DATASET	Amount of log files	Size (GB)
A	121 733	15.4
B	153 874	19.5
TOTAL	275 605	34.9
SUBSET	**Amount of log files**	**Size (GB)**
A	5040	0.659
B	4000	0.514
TOTAL	9040	1.17

7. First Iteration: Using the *logparser* Toolkit

The first iteration included the usage and evaluation of the *logparser* (P. He et al., 2016; Zhu et al., 2019) and *loglizer* (S. He et al., 2016) toolkits. The design of the artefact was based on this investigation, and the existing tools were used in practice in order to see how they could solve the problem. In addition, the performance and the utility of the tools were evaluated. Thus, the first iteration encompassed steps 3 (design and development), 4 (demonstration), and 5 (evaluation) of the Design Science Research process model (Peffers et al., 2007).

This section describes the usage of the *logparser* toolkit, and is organised as follows. Sections 7.1, 7.2 and 7.3 describe the tests performed with the tools LenMa, SHISO, and AEL, respectively. Parsing the entire data subset into structured data is explained in section 7.4.

The three tools from the *logparser* toolkit which achieved the highest accuracy with Linux log files in Zhu et al. (2019) were tested by parsing the same single log file with each tool. The test log file contained 1597 log messages, and used 111 KB of disk space. The tools were LenMa (Shima, 2015), SHISO (Mizutani, 2013), and AEL (Jiang et al., 2008). The tools can be optimized with regular expressions, and two expressions which help the tools in handling IP addresses (e.g. "100.55.123.1") and time information (e.g. "08:15:00") were used with all of them.

7.1 Testing LenMa

LenMa is short for "Length Matters", and its event template extraction is based on the token length properties of log messages, as the name suggests (Shima, 2015). When the test file was parsed with LenMa, the tool was able to determine 1018 different event templates, meaning the reduction in relation to the log entries in the file was only 36.26%.

Inspecting the resulting event templates revealed that the accuracy of LenMa was quite disappointing. For example, the log file contained numerous entries about some settings of the device, presented as key-value pairs in the style of "*setting: VALUE*". LenMa determined that each of these pairs is an individual event template, even though the entries clearly have a constant part "*setting:*".

In order to mitigate this issue, two additional, complex regular expressions were used. One expression helped the tool in finding the variables in the "*setting*:" strings and the other did the same for "Item #0 None" style strings, which were similarly challenging for LenMa. With the regular expressions, LenMa performed better, extracting 813 event templates. A large amount (677 templates, 83% of total) of these occurred only once, which may indicate that quite a few events were incorrectly determined to be individual instances, instead of belonging to a single template. In fact, the output contained ten templates that started with the words "inserted module", followed by a word varying across the templates, i.e. a variable, for example.

7.2 Testing SHISO

SHISO generates nodes from log messages, creating a structured tree, and is capable of refining the log format continuously in real time (Mizutani, 2013). The initial parsing test immediately showed much promise, compared to LenMa, as SHISO extracted 611 event templates from the test log (a 61.74% reduction from the amount of log entries). This result was achieved with only the two basic regular expressions for IP addresses and time information.

Closer inspection showed that the situations which caused trouble for LenMa were handled with ease by SHISO. For example, regarding the "*setting: VALUE*" entries, SHISO correctly determined that such entries can be encompassed in a single event template. However, SHISO extracted all-variable event templates, resulting in templates such as "* *" or "* * * * *", which could in theory match any two- or five-word log entry. This is obviously undesirable, since for example log entries "*Everything is fine*" and "*Device will explode*" both contain three words, and would fit the template "* * *", but clearly cannot be considered as two instances of the same event. SHISO determined that all-variable events occurred 312 times in the log file.

7.3 Testing AEL

The log parsing process of AEL includes four steps: anonymize, tokenize, categorize, and reconcile (Jiang et al., 2008). The original algorithm merges events in the reconcile step, but cannot handle cases where a single template has multiple different parameter tokens (Logpai, 2018). To address this issue, the algorithm is improved in the *logparser* implementation (Logpai, 2018).

Like SHISO, AEL showed promise in its parsing accuracy. 591 event templates were extracted, again with only the two basic regular expressions. Based on the smaller amount of event templates, it can be considered that AEL performed even better than SHISO. Moreover, AEL did not extract any all-variable templates, outperforming SHISO in this regard as well.

7.4 Parsing the subset

The performance of the three tested tools was, surprisingly, completely contrary to what the measurements in Zhu et al. (2019) would suggest. That is, even though LenMa was measured to be the most accurate of the three, it seems to perform the worst here, for example. Since SHISO and AEL were much more accurate than LenMa, even without additional regular expressions, only they were used for the remaining log parsing tasks.

The data subset described in section 5 was thus parsed with both SHISO and AEL. Having two different versions of the parsed logs made it possible to compare the accuracy of the two tools from the viewpoint of the anomaly detection tools: if the anomaly detection is considerably more accurate with logs parsed by one tool, that tool is most likely more accurate in its log parsing. Likewise, the effect of the parsing accuracy on the performance of the anomaly detection was investigated.

The subset included about a thousand log files from each device, so the log parsing was done in cycles of 1000 log files. An interesting observation was immediately made: AEL is much faster than SHISO. On average, AEL parsed 1000 log files in about 4

minutes 46 seconds, whereas SHISO processed for 2 hours, 6 minutes and 36 seconds. In this context, when the objective is to generate large amounts of structured logs, SHISO is thus far less useful. However, the typical use case of the anomaly detection tool only includes the inspection of a single log file. In that context, the user has to wait for SHISO to complete for only about seven or eight seconds. Of course, this is still slow compared to AEL, which only needed as little as 0.2 seconds for a single log file, but not unreasonably slow. Nevertheless, based on these tests, as AEL is better in both parsing accuracy and processing time, I would strongly recommend it for parsing Linux logs.

The resulting structured logs were stored as .csv (comma-separated values) files. The values extracted from each log entry were LineId (line number of the log entry), Month, Date, Time, Level (name of the device, always the same), Component (software component which logged the entry), PID (process identification number), Content (free-form textual log message), EventId (identification for the log event), and EventTemplate (the constant part of the content).

8. First Iteration: Using the *loglizer* Toolkit

This section describes the background and usage of *sgologs_alpha*, the first version of *sgologs*. The main purpose of *sgologs_alpha* was running *loglizer* with different machine learning models and different kinds of log sequences as efficiently as possible. *Sgologs_alpha* does not include *logparser* tools – instead, the input has to be a previously parsed, structured log file. Other than that, the functionality is quite the same as with the final *sgologs*. *Sgologs_alpha* is a straightforward command-line tool, with arguments such as the input log file and optionally the machine learning model to use. The user can also specify the basis of splitting the input log into sequences, i.e. Time or PID. Training data, which can also be given as a command-line argument, is then loaded. After the machine learning model is initialized and trained, anomalies are detected from the input log file. The process *sgologs_alpha* goes through is illustrated in Figure 4.

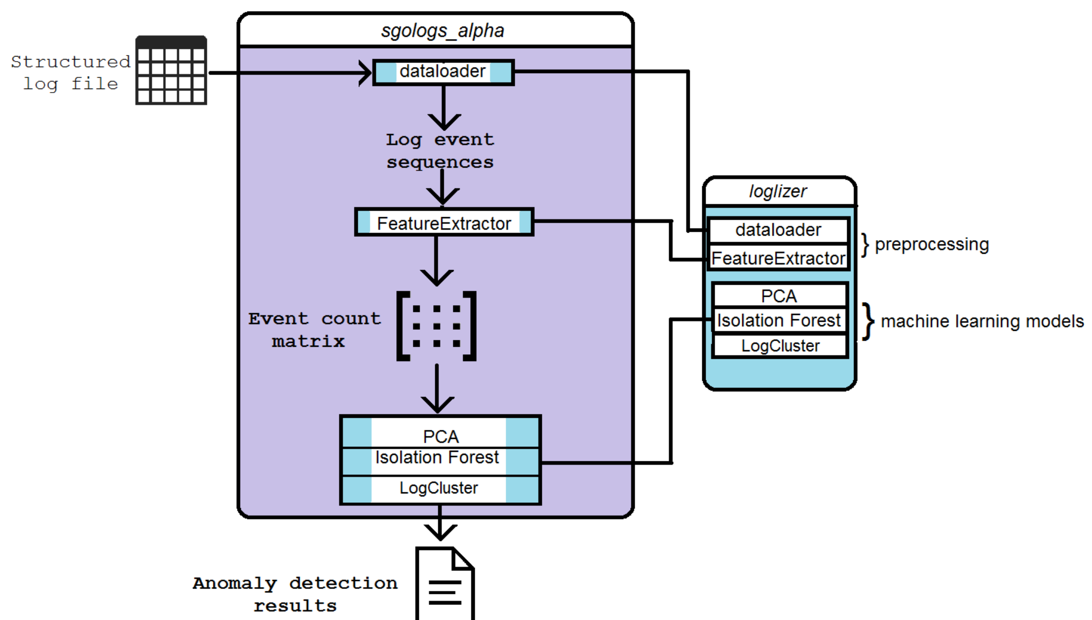

Figure 4. The anomaly detection process of *sgologs_alpha* with different parts of *loglizer*.

The remainder of this section is organized as follows. Arranging the log events into sequences is discussed in section 8.1. Then, the used training and test data and its preparation are presented in section 8.2. In section 8.3 a walkthrough of *loglizer* is provided, in order to illustrate how the software works and how the anomaly detection framework (S. He et al., 2016) is used in practice. Section 8.4 presents the anomaly detection results achieved with *sgologs_alpha*. Finally, section 8.5 summarizes the first iteration.

8.1 Sequence determination

Loglizer does not detect anomalies at the level on individual log messages. Instead, messages are grouped into log event sequences, and the machine learning classifies each sequence as normal or anomalous. S. He et al. (2016) determined their log sequences with fixed windows, sliding windows, and session windows. Fixed and sliding windows were based on timestamp data, but the timestamps in the Linux dataset are to a large extent arbitrary – for example, the time may suddenly shift by several hours between consecutive log entries. Using the timestamps as basis for fixed and sliding windows was not convenient. Instead, timestamps were used as identifiers for session windows, along with the process identification number (PID) assigned to each log entry. How log sequences are created with different window types and settings used in this study is illustrated in Figure 5. The log extract is an artificial Linux log, generated for figurative purposes only.

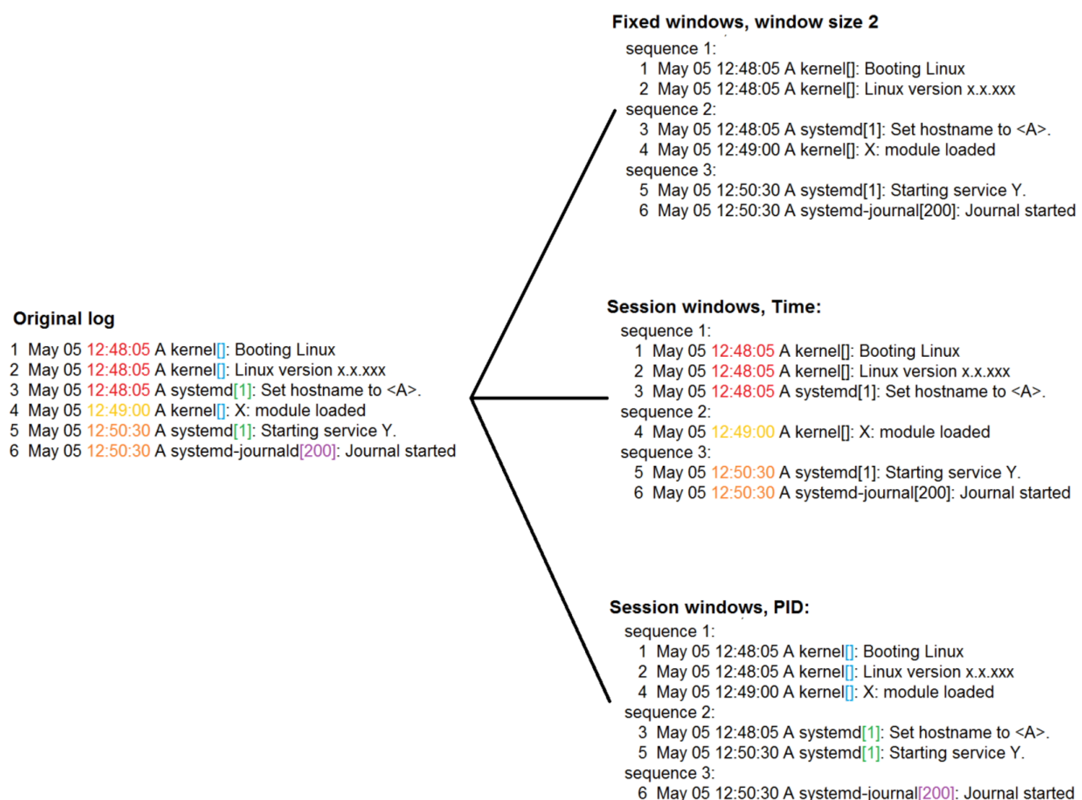

Figure 5. Illustration of how different kinds of log sequences are formed.

As shown in the Figure, each window type produces rather unique log event sequences. Fixed windows are the only sure method of generating sequences with equal length, determined by the window size. For example, if the window size was set to four in the example in Figure 5, the first sequence would naturally include log events one, two, three, and four, whereas events five and six would belong to the second sequence. It is also noteworthy that session windows based on PID values do not retain the order of events in the original log, as event four is included in the first sequence but event three in the second.

Implementing fixed or sliding windows based on the amount of log entries is a possibility. Splitting the log entries into sequences that contain some predefined amount of entries, e.g. as in Mäkinen (2019), is obviously a convenient approach. However, it is

possible that the selected amount is simply an arbitrary value, resulting in sequences that may not accurately represent the actual process sequences of the system. One advantage of this approach is that the sequences are enforced to be equal in length. This approach was implemented during the second iteration of the development process, and during the first iteration focus was restricted to session windows based on values in the Time and PID fields. The implementation was delayed because the existing implementations relied on label data, and creating a label file independent approach required additional effort.

Common to all session windows generated with the two different identifiers is that sequence length varies considerably. For example, considering timestamps, one log file contained 373 log entries with the timestamp 07:44:35, whereas the amount of entries with the timestamp 07:44:37 was only two. Sequences generated from the PID values ranged from one to more than four hundred in length. Varying sequence lengths did not prevent the machine learning from detecting real anomalies, but Time sequences were much more suitable for the anomaly detection than PID sequences, as discussed in section 8.4.

8.2 Training and test data

Initially, training data was constructed from a thousand log files. Their contents were grouped into 56 761 event sequences, using session windows based on Time values. The thousand files were randomly selected, and the training data was assumed to be large enough to represent the data as a whole. The typical use case of *sgologs* is running anomaly detection on a single log file, which is why a test file was selected. The structured test file is very long, over 600 KB, whereas a log file from the dataset parsed with *logparser* is normally just over 200 KB in size. It is possible that the test file is somehow corrupted. It is also notable that the test file includes 1 362 log entries only from the kernel, although the log files commonly have about 1 750 entries in total. The main purpose of using this test file was investigating the suitability of the training data.

Typically, especially in supervised machine learning, the data is split to training-, validation and test datasets in some proportion, e.g. 50:25:25 (Pietikäinen & Silvén, 2019). However, such a division was not purposeful in this context. Machine learning models are usually trained once, and they store their learned parameters for future use. *Loglizer*, however, is not this sophisticated. As a peculiar feature, the models need to be trained each time the software is run. Therefore the user has to wait for the training phase to complete before the anomaly detection begins and results become available. Consequently, a large training dataset which takes a long time to process limits the utility of the tool, as running anomaly detection on a single log file becomes e.g. a 20-minute task.

The first tests quickly revealed that a training dataset generated from one thousand log files is already too large. The `fit_transform` function processed for more than ten minutes, and the model training (with `model.fit`) was even more time-consuming. Therefore, a training dataset of 4520 log files makes no sense. Moreover, as mentioned above, *sgologs* is typically supposed to be run with a single log file, making a pool of thousands of test files equally unsuitable.

Finding a suitable size for the training data took some time. The processing time for training data of 500 log files was still well over ten minutes, which is unacceptably long for a user-friendly tool. Reducing the amount of log files by half again improved the

situation, and the total processing time was a little over one minute. Such a processing time is quite acceptable, but the amount was ultimately reduced to 200 to enable an even more fluent user experience. The processing time for 200 log files was 43 seconds.

200 is of course a lot less than 1000, but 200 logs still contain more than 11 000 log event sequences. Moreover, should the user worry whether this is enough, they have the option of specifying which training data *sgologs* uses. If the user wants a second opinion, so to speak, after inspecting the output, they can run *sgologs* again with another training dataset.

The output of *sgologs* using the PCA model on the test file was the exact same with all training data sizes. Five anomalous sequences were detected: the first two were unconventionally long sequences, including the kernel sequence mentioned above, and the remaining anomalies were three separate sequences with the same content. Log sequences of the training data and the test file were generated with session windows, using the process identification numbers (PID).

Final test files, which were not included in the training datasets, were taken from the A directory. The files were selected based on the amount of the word *error* in them. Log files with few errors, a normal amount of errors, and many errors, were all represented. Log messages with “error” are of course easy to find even without anomaly detection. However, as every boot of the Linux system in question produces at least some error messages in the log, without anomaly detection the developer would have to be able to recognize or recall which errors are “normal.” Moreover, focusing on errors simplifies the validation of the results. As the generation of comprehensive label files, i.e. classification data on all kinds of sequences in the test files, was beyond the scope of this study, it was appropriate to focus on “obvious” anomalies, such as additional errors.

As mentioned earlier, the log files commonly have about 1750 log entries, and the word *error* typically occurs 16 times. One such file, 174_3_11, which has 1762 rows and 16 occurrences of *error*, was included to see how the anomaly detection reacts to “normal” log files. The file 174_3_7 included 22 occurrences of *error* and 2765 rows, and was selected as an example of a log with many errors, which supposedly leads to many detected anomalies. On the other end of the scale is the file 174_3_18, which contains 1563 rows and only 5 occurrences of *error*. It seems that the log ends too soon, i.e. a few hundred log entries are missing from the end of the file. As small amount of errors as possible is obviously desirable for any software, but in this context the machine learning should determine such behaviour as anomalous, since the amount of errors is typically almost four times larger. Therefore it was interesting to see what *sgologs* does with 174_3_18. The three test files are summarized in Table 4.

Table 4. Test files, their size, and amount of *error*.

Test file	Amount of log entries	Occurrences of *error*
174_3_7	2 765	22
174_3_11	1 762	16
174_3_18	1 563	5

A typical log file, e.g. 174_3_11, includes two error messages from a sound server component called PulseAudio. Both messages state that PulseAudio was “unable to contact D-Bus,” and actions other than autolaunch should be executed. In 174_3_7, this pair is repeated three additional times, resulting in the six additional occurrences of *error*. This produces two or four anomalous sequences, depending on how the log event sequences were generated. The eleven occurrences of *error* missing from 174_3_18 but

present in 174_3_11 are log messages from CherryPy, a Python web framework. The messages are in fact not errors, and also feature the keyword "INFO", but for some reason the CherryPy message format seems to always be "cherrypy.error: <level>: <message>". It is assumed that the sequence that would precede the missing data, i.e. the last sequence in 174_3_18, should be classified as an anomaly.

8.3 A walkthrough of *loglizer*

As mentioned in section 4, the anomaly detection process framework specified by S. He et al. (2016) has four steps: log collection, log parsing, feature extraction, and anomaly detection. *Loglizer* conducts steps three and four. *Loglizer* includes demo files, simple Python scripts that demonstrate the usage of the API. The following code sample is extracted from the file "InvariantsMiner_demo_without_labels.py" and slightly simplified.

```
from loglizer.models import InvariantsMiner
from loglizer import dataloader, preprocessing

struct_log = '../data/HDFS/HDFS_100k.log_structured.csv'
# The structured log file
epsilon = 0.5 # threshold for estimating invariant space

if __name__ == '__main__':
    # Load structured log without label info
    (x_train, _), (x_test, _), d_fr = dataloader.load_HDFS(struct_log,
                                                     window='session',
                                                     train_ratio=0.5)
    # Feature extraction
    feature_extractor = preprocessing.FeatureExtractor()
    x_train = feature_extractor.fit_transform(x_train)

    # Model initialization and training
    model = InvariantsMiner(epsilon=epsilon)
    model.fit(x_train)

    # Predict anomalies on the test set to simulate the online mode
    # x_test may be loaded from another log file
    x_test = feature_extractor.transform(x_test)
    y_test = model.predict(x_test)
```

Loglizer has a ready implementation for HDFS log files, and a partial implementation for BGL logs, both of which assume that label data is available. These implementations reside in the `dataloader` module, which takes care of reading the structured log file(s) and arranging the log entries into sequences. By default, the log is also split in half, and the first half is used as training data and the second as test data in the demo files. The `fit_transform` function from the `FeatureExtractor` class turns the training data sequences into an event count matrix, where each row represents a log sequence and consists of aggregated event counts. As the class name suggests, this activity belongs to the third step of the anomaly detection framework. S. He et al. (2016) also include the sequence determination, performed by the `dataloader` module, in the step of feature extraction.

The `epsilon` value is an example of a parameter which might require tuning depending on the used data. The `transform` function transforms the test data into an event count matrix, using parameters acquired in the `fit_transform` function. The `predict` function of the machine learning model (Invariants miner in this case)

returns the classification of each log sequence; 0 stands for normal, and 1 indicates the sequence is anomalous. Using the `fit` and `predict` functions composes the fourth step of S. He et al.'s (2016) framework, anomaly detection.

In order for *loglizer* to work on Linux logs, a `load_Linux` function was implemented in the `dataloader` module. The function differs from the existing `load_HDFS` in three ways. First, the function extracts the log sequences based on values that appear in Linux logs, instead of block ID values in HDFS logs. Sequence determination is described in 8.1. Second, all if-branches that are triggered when label data is available are removed. Third, the function returns an additional data frame. The original `load_HDFS` returns the event sequences as arrays, split to training and test data, and all the event sequences as a pandas[3] data frame. Such a data frame can be seen in Figure 6.

```
      PID                                       EventSequence
0          [96dfbadc, 74adcd00, 4eb964bd, 4bfae480, ad5a3...
1       1  [9178bfc3, 408e3f56, a458faaf, 19a1fc1c, 12d15...
2     227                      [9615c27f, 3523023f, 3523023f]
3     221  [e2d09e15, e2d09e15, e2d09e15, e2d09e15, e2d09...
4     236  [a5b966ca, a4b9ffee, de202140, de202140, 89ed9...
..    ...                                                 ...
151  4429            [23526083, 23526083, 23526083, 23526083]
152  4221  [c4f14310, c6098763, d5b45273, b0818b3d, d5b45...
153  4227  [eb9d3c6c, 42249166, 259c6ad0, cd9cc16d, 7136a...
154  4844                      [b9b67684, 67ba990f, aeebac42]
155  4932                                          [b9b67684]

[156 rows x 2 columns]
```

Figure 6. An example of a pandas data frame of log event sequences.

In the data frame visible in Figure 6 the event sequences are generated with session windows based on PID values. Each log entry that has the same PID value is grouped into the same event sequence. For example, the log entries with the event IDs '9615c27f' and '3523023f' both have the value 227 in their PID field, and event '3523023f' occurs twice. The additional data frame generated in `load_Linux` is similar, but the event sequences consist of the actual log messages, instead of the event IDs. Thus, it is easier to present a meaningful output of the anomalous sequences.

An output of *sgologs_alpha* on the command prompt can be seen in Figure 7. Most of the text following the tool's name, the test log file, and the used machine learning model, are default prints from *loglizer*.

[3] pandas is a popular Python data analysis library. https://pandas.pydata.org/

```
*******
SGOLOGS
*******
Anomaly detection for ../../data/test/AEL/174_20191108_3_7_structured.csv
with PCA
***Feature extraction***
====== Transformed train data summary ======
Train data shape: 6016-by-2361

***Model initialization and training***
====== Model summary ======
n_components: 2
Project matrix shape: 2361-by-2361
SPE threshold: 1400.0

***Predict anomalies***
====== Input data summary ======
Loading ../../data/test/AEL/174_20191108_3_7_structured.csv
Total: 30 instances, train: 0 instances, test: 30 instances
====== Transformed test data summary ======
Test data shape: 30-by-2361

Anomalous sequences found:
        Time                                      EventSequence
0   15:01:16  [96dfbadc, 74adcd00, 4eb964bd, 4bfae480, ad5a3...
6   04:04:43  [f1f69ecf, de85557c, ed2d9788, a2edb758, f9a5f...
13  04:04:50  [9545e015, 9545e015, 9545e015, 9545e015, 9545e...
15  04:04:52  [650a0d79, 50188105, a8bfee1c, 3535e714, 3c876...
Results in sgologs_res/sgologs_1_pca/anomalies_eID_seq.csv & anomalies_content_seq.txt
Time taken: 0:00:25.377512
```

Figure 7. An example output of *sgologs_alpha* using AEL-parsed logs and PCA

The “shape” of the training and test data refers to the dimensions of the event count matrix. The “Model summary” shows specific information about the model and its parameters. The value for SPE threshold was manually set to 1400.0, as discussed in section 8.4.1. In this case, the model classified some sequences as anomalies, and they are shown in the output as lists of event IDs. *Sgologs_alpha* also notifies that more verbose output, including the actual message contents of the anomalous sequences, can be found in the files *anomalies_eID_seq.csv* and *anomalies_content_seq.txt*.

8.4 Evaluation of *sgologs_alpha*

This section describes the results *sgologs_alpha* achieved with the training data and the three test files. Log sequences were determined with session windows based on Time and PID values in the log entries. Log files were parsed with both SHISO and AEL from the *logparser* toolkit. The output of *sgologs_alpha* with Time sequences and SHISO logs is discussed in section 8.4.1, with Time sequences and AEL logs in 8.4.2, with PID sequences and SHISO logs in 8.4.3, and with PID sequences and AEL logs in 8.4.4.

8.4.1 Using Time sequences and SHISO logs

Generating session windows from the Time values resulted in 32 log event sequences in 174_3_11, 30 sequences in 174_3_7, and 26 sequences in 174_3_18. As mentioned earlier, their lengths varied between 1 and over 400. The test logs parsed with SHISO were processed first. Naturally, the training data was also parsed with SHISO.

The PCA model definitely suffered from a high amount of false positives. For example, in 174_3_7, PCA determined that 18 sequences are anomalous. These sequences contained 21 occurrences of error (out of 22), which is not surprising since a majority of the total event sequences were classified as anomalous. The utility of such an output is questionable, since the list of log entries included in the anomalous sequences is essentially the original log file reduced by a few hundred rows.

Invariants Miner suffered from an exceedingly long processing time. *Sgologs_alpha* was run with Invariants Miner numerous times, with different parameters, but each time the processing time approached ten minutes. The situation was the same with all the test files and using log files parsed with AEL had no effect. This finding was disappointing, since Invariants Mining was measured to be the best unsupervised machine learning model in S. He et al. (2016). Nevertheless, due to the unacceptable processing time, using Invariants Miner had to be discontinued at this point.

LogCluster initially showed more promise than PCA. The model found one anomaly from 174_3_11, the "normal" log file, when comparing to the training data from the A directory, and two anomalies when compared to B training data. These anomalies did not contain any occurrences of *error*, but on the other hand the typical amount of 16 errors should not even be considered anomalous. The file with more errors, 174_3_7, was determined to have four (A training data) or three (B training data) anomalous sequences. These sequences encompassed a half of the total amount of *error*. However, these *errors* are the eleven CherryPy messages that should appear in every log. The additional PulseAudio errors, i.e. the real anomalies, were not included in the anomalous sequences.

LogCluster detected two or four anomalous sequences in 174_3_18, depending on the training data. The second-to-last sequence was classified anomalous each time. One could suppose that the machine learning expects that the sequences near the end would be followed by the few hundred entries missing from 174_3_18, and thus classifies the final sequences as anomalous. However, it is unclear why the last sequence was considered normal. LogCluster's processing time, 1 minute 5 seconds on average, was a little longer than PCA's (about 45 seconds).

The processing time of Isolation Forest was even shorter, about 42 seconds on average. However, for all the log files and with both training datasets, Isolation Forest always classified the first sequence as anomalous and everything else as normal. The results were the exact same with logs parsed by AEL as well; the only thing that changed was the processing time. It is difficult to believe that this would be just a coincidence, especially since there is nothing particularly strange in the first sequences in any of the test files. It is justified to conclude that the results provided by Isolation Forest are false.

At this point it was apparent that the machine learning models needed some fine-tuning. Each model implementation included initialization parameters, for example a threshold for anomaly detection, which can be set by the user. The above results were achieved with default parameters, or parameters automatically calculated by the models. For instance, PCA calculated a very low threshold value for the anomaly detection, and increasing this value greatly reduced the number of false positives. Isolation Forest needed an estimation of anomaly samples in the data, and tuning that parameter resulted in much more sensible output. LogCluster had thresholds for clustering and anomaly detection, but non-default values for these parameters only made the results less accurate. Therefore the default parameters were used for LogCluster.

Some of the results achieved by the models are presented in the following Tables. Using the new parameters, the results of PCA are presented in Table 5 and the results of Isolation Forest in Table 6. In all the Tables in section 8 the ideal situation would be that Real anomalous sequences, Detected anomalous sequences, and True positives have the same value, and False positives (normal sequences classified as anomalies) and False negatives (undetected real anomalies) are both zero. This would require that the machine learning achieves perfect anomaly detection accuracy, which is unrealistic.

Table 5. Anomaly detection results of PCA with Time sequences and SHISO log files

Test file (training data)	Real anomalous sequences	Detected anomalous sequences	True positives	False positives	False negatives
174_3_7 (A_170)	2	4	1	3	1
174_3_7 (B_178)	2	4	1	3	1
174_3_11 (A_170)	-	4	-	4	-
174_3_11 (B_178)	-	4	-	4	-
174_3_18 (A_170)	1	2	0	2	1
174_3_18 (B_178)	1	2	0	2	1

Table 6. Anomaly detection results of Isolation Forest with Time sequences and SHISO log files

Test file (training data)	Real anomalous sequences	Detected anomalous sequences	True positives	False positives	False negatives
174_3_7 (A_170)	2	6	1	5	1
174_3_7 (B_178)	2	7	0	7	2
174_3_11 (A_170)	-	7	-	7	-
174_3_11 (B_178)	-	6	-	6	-
174_3_18 (A_170)	1	7	0	7	1
174_3_18 (B_178)	1	7	0	7	1

The processing times of the models were reduced with the new parameters. The average processing times for PCA and Isolation Forest were 43.5 and 40.5 seconds, respectively. PCA's results were much better, as only two to four sequences were classified anomalous. Moreover, a real anomalous sequence in 174_3_7 was detected. The real anomaly in 174_3_18 was not detected. It seems that PCA is not greatly affected by the training data, as the results are usually the exact same with A_170 training data as with B_178 training data.

Isolation Forest produced a much more practical output with the new parameters. The model classified varied sets of sequences as anomalies, and found a real anomaly in 174_3_7 with A_170 training data. Overall accuracy was not as good as with PCA, since Isolation Forest did not detect real anomalies with B_178 training data, and the rate of false positives was higher.

The commonly used metrics precision, recall and F-measure are used to represent the accuracy of machine learning models. As mentioned in section 4.2.3, precision is the percentage of how many detected anomalies are correct, recall is the percentage of how many real anomalies are detected, and F-measure is the harmonic mean of the two (S. He et al., 2016). Higher precision indicates fewer false positives, and higher recall indicates fewer false negatives. Figure 8 shows the precision, recall, and F-measure scores of the models, when detecting anomalies from the 174_3_7 test file.

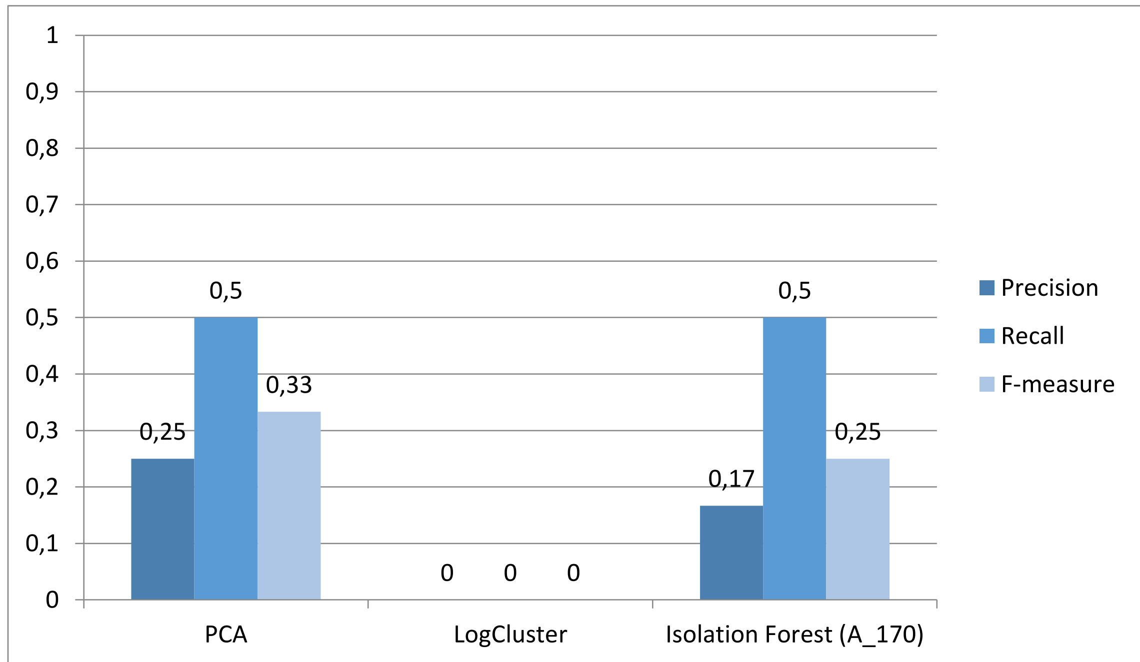

Figure 8. ML model scores with Time sequences and SHISO logs on file 174_3_7

It should be noted again that only the anomalies of additional PulseAudio error messages is taken into consideration. In Figure 8, the best results of the models are shown; Isolation Forest achieved the best accuracy with the training dataset A_170. Since PCA and Isolation Forest both found one of the two anomalies, the recall is exactly $\frac{1}{2} = 0.5$. The precision scores are quite low, since the models classified quite many false positives. As LogCuster's section in the Figure reveals, if the amount of true positives is zero, so are the precision, recall and F-measure scores.

If the F-measure scores are compared to the scores measured in S. He et al. (2016), the accuracy achieved here seems rather substandard. For instance, the worst F-measure score for PCA in S. He et al. (2016) was 0.55. When the number of real anomalies is low, such as two, even one false negative greatly reduces the recall score. Similarly, false positives, common in unsupervised learning (Landauer et al., 2018), reduce the precision score, and appear to be the main reason of a reduced F-measure score.

8.4.2 Using Time sequences and AEL logs

The results were quite different when log files parsed with AEL were used. The accuracy of PCA remained, and the processing time was reduced to 25.5 seconds on average. The amount of false positives with training data B_178 was unfortunately increased.

LogCluster found fewer anomalies with AEL-parsed files than with SHISO log files. Overall, the anomalous sequences were almost the same, except that only one sequence was classified anomalous in 174_3_18. Real anomalous sequences were not detected. LogCluster came closest with 174_3_7 and A_170 training data, as a sequence immediately following the anomalous sequences was detected. Processing AEL files was faster with LogCluster as well, as the average processing time was about 50 seconds. A major downside was that sequences classified as anomalies did not contain any occurrences of *error*.

Isolation Forest improved its performance, as real anomalies in 174_3_7 were detected with both training datasets. It is also notable that Isolation Forest correctly classified both sequences in 174_3_7 which could be considered anomalous. Average processing time was 28 seconds.

Tables 7 and 8 present the results using log files parsed with AEL. LogCluster results are in Table 7, and Isolation Forest results are in Table 8.

Table 7. Anomaly detection results of LogCluster with Time sequences and AEL log files

Test file (training data)	Real anomalous sequences	Detected anomalous sequences	True positives	False positives	False negatives
174_3_7 (A_170)	2	3	0	3	2
174_3_7 (B_178)	2	2	0	2	2
174_3_11 (A_170)	-	2	-	2	-
174_3_11 (B_178)	-	1	-	1	-
174_3_18 (A_170)	1	1	0	1	1
174_3_18 (B_178)	1	1	0	1	1

Table 8. Anomaly detection results of Isolation Forest with Time sequences and AEL log files

Test file (training data)	Real anomalous sequences	Detected anomalous sequences	True positives	False positives	False negatives
174_3_7 (A_170)	2	5	2	3	0
174_3_7 (B_178)	2	5	2	3	0
174_3_11 (A_170)	-	5	-	5	-
174_3_11 (B_178)	-	5	-	5	-
174_3_18 (A_170)	1	5	0	5	1
174_3_18 (B_178)	1	4	0	4	1

Anomaly detection seems to perform better with AEL-parsed logs. PCA and Isolation Forest both find real anomalies in the 174_3_7 file, independent of the used training data. A reduced processing time, compared to log files parsed with SHISO, is also a significant advantage. LogCluster appears quite useless, at least with sequences based

on Time values. Figure 9 shows the precision, recall, and F-measure scores with AEL-parsed training data and 174_3_7 test file.

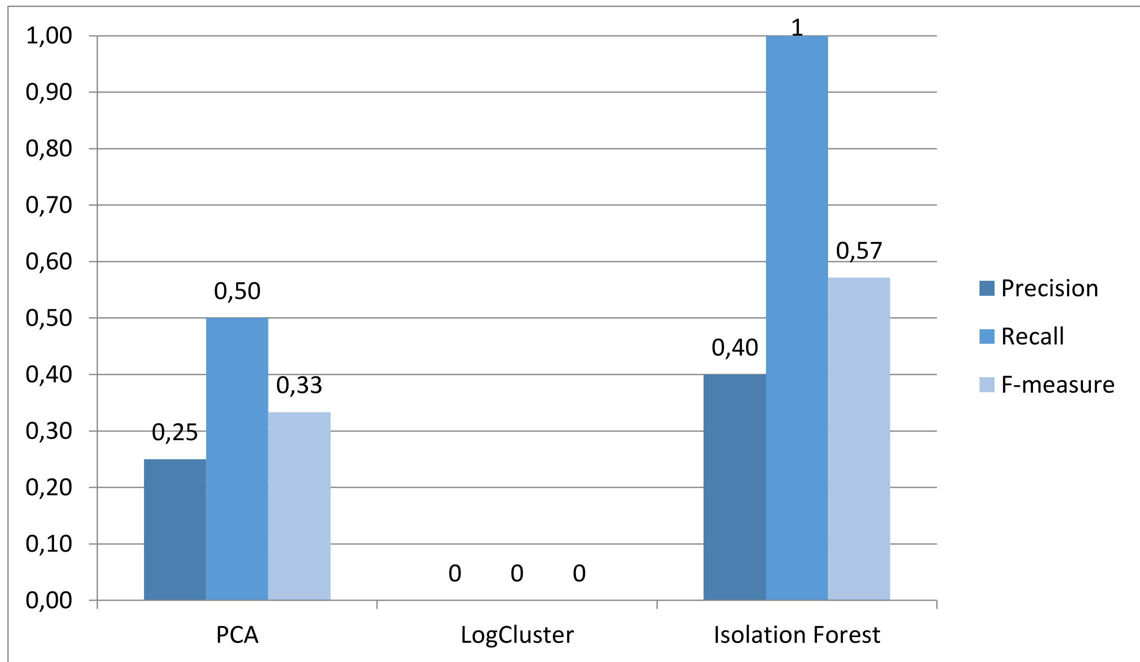

Figure 9. ML model scores with Time sequences and AEL logs on file 174_3_7

As can be seen in the Figure, the performance of PCA and LogCluster are at the same level as with SHISO log files. However, Isolation Forest achieved much better scores, regardless of the used training data. All real anomalies were detected, resulting in the highest possible recall score, which also had a positive effect on the F-measure. In this study, high recall is better than high precision, since many false negatives are considered to be more severe than many false positives.

8.4.3 Using PID sequences and SHISO logs

The training data with PID sequences was generated from the same log files as the training data used in the previous sections. Test file 174_3_7 has 63 PID sequences, 174_3_11 has 54, and 174_3_18 has 52.

One advantage of the Time sequences is that all sequences and all log events in the sequences actually occurred in that specific order. However, processes, i.e. PID sequences, occur concurrently. Thus, different processes overlap in the log files, and different log entries with the same PID value are not necessarily successive. For this reason it is impossible to determine if some sequence should follow another, or if some sequence is missing. If the CherryPy functions were not run in a separate process, with its own PID value, it would be possible to recognize their absence in 174_3_18 based on the finding that some PID sequence is abnormally short. However, in the present situation, detecting anomalies in 174_3_18 using PID sequences is not possible.

Anomaly detection with PID sequences was performed for 174_3_7 and 174_3_11 only. Tables 9 and 10 present the results for PCA and Isolation Forest, respectively.

Table 9. Anomaly detection results of PCA with PID sequences and SHISO log files

Test file (training data)	Real anomalous sequences	Detected anomalous sequences	True positives	False positives	False negatives
174_3_7 (A_170)	4	5	0	5	4
174_3_7 (B_178)	4	5	0	5	4
174_3_11 (A_170)	-	4	-	4	-
174_3_11 (B_178)	-	4	-	4	-

Table 10. Anomaly detection results of Isolation Forest with PID sequences and SHISO log files

Test file (training data)	Real anomalous sequences	Detected anomalous sequences	True positives	False positives	False negatives
174_3_7 (A_170)	4	8	0	8	4
174_3_7 (B_178)	4	8	0	8	4
174_3_11 (A_170)	-	5	-	5	-
174_3_11 (B_178)	-	5	-	5	-

As can be seen from the Tables, the performance of the machine learning models was significantly worse with PID sequences. Tuning the parameters again did not have any positive effect on the results. As PID sequences were more numerous than Time sequences, processing times were also much higher: on average 1:08 with PCA and LogCluster and 1:21 with Isolation Forest.

With PID sequences and SHISO logs, PCA and Isolation Forest were unable to detect real anomalies in 174_3_7. LogCluster performed similarly as with Time sequences, without detecting the anomalies in 174_3_7. The only errors in the outputs were the normal CherryPy messages, which LogCluster incorrectly detected in 174_3_7 and 174_3_11. The amount of false positives classified by PCA and Isolation Forest was also high. As the amount of true positives in 174_3_7 was zero with all the models, the precision, recall and F-measure scores were also zero. It is quite clear that anomaly detection from log files parsed with SHISO should not be done with sequences based on PID values.

8.4.4 Using PID sequences and AEL logs

Changing from SHISO logs to logs parsed with AEL did not improve the anomaly detection accuracy when using PID sequences. The only difference was significantly reduced processing times. The average processing times were as follows: PCA: 40 seconds, LogCluster: 37 seconds, Isolation Forest: 51 seconds. Precision, recall and F-measure were still zero with all models, and the real anomalies in 174_3_7 were not detected. A peculiar finding was that two sequences in 174_3_11 were consistently classified as anomalies by PCA and Isolation Forest: sequence 1, a systemd process, and sequence 18, a SOC4E process. These classifications were made with both SHISO and AEL log files. However, there does not seem to be anything particularly anomalous in the sequences. The results achieved with LogCluster are presented in Table 11. With

these parameters, LogCluster's results could be considered the best, since the amount of false positives was the lowest. Of course, true positives are still zero, as with the other models.

Table 11. Anomaly detection results of LogCluster with PID sequences and AEL log files

Test file (training data)	Real anomalous sequences	Detected anomalous sequences	True positives	False positives	False negatives
174_3_7 (A_170)	4	2	0	2	4
174_3_7 (B_178)	4	2	0	2	4
174_3_11 (A_170)	-	2	-	2	-
174_3_11 (B_178)	-	3	-	3	-

Session windows based on PID values appear to be unsuitable for anomaly detection with these Linux log files. The machine learning models apparently benefit from sequences which retain the actual order of the log events, such as the session windows based on values in the Time field of the log entries.

8.5 Summary of the first iteration

AEL is quite clearly the best tool for these log files in the *logparser* toolkit. Compared to SHISO, AEL's processing time is substantially shorter, and the parsing accuracy is considerably better. Moreover, the processing times of *loglizer* tools are also shorter with AEL-parsed logs, and the anomaly detection accuracy is increased if AEL logs are used instead of SHISO logs.

How the log files are split and organized into sequences in the process is not a trivial matter. The models were unable to detect the real anomalies in the test files when using PID sequences, thus session windows based on PID values in the log entries are not suitable in this context. Sequences which retain the occurrence order of the log entries as well as the order of the sequences themselves enable the best anomaly detection accuracy. Such sequences include session windows based on Time values, and fixed windows, which were implemented during the second iteration and are discussed in section 9. This finding is quite contrary to what S. He et al. (2016) discovered: session windows used with HDFS logs resulted in higher correlation between events in each sequence, which is why anomaly detection methods perform better than with other log data. HDFS logs are "easy" for the machine learning models also because the amount of different event types is only 29 (S. He et al., 2016). As described in section 7, the amount of unique event templates in a single log file used in this study can more than 500.

As for the machine learning models in charge of the anomaly detection, Invariants Miner and LogCluster are unsuitable for this task. The processing time of Invariants Miner is close to a small eternity, which is unacceptable for a tool which is supposed to be fast and convenient to use. In fact, S. He et al. (2016) mention that the invariants mining process is time consuming. If *loglizer* did not need training on each run, a prolonged training time would not be an issue.

LogCluster's processing time is also longer than that of PCA or Isolation Forest, but in addition the detection accuracy of LogCluster is virtually non-existent. S. He et al. (2016) also found that LogCluster does not obtain a good accuracy on BGL data. The characteristics of the generated event count matrix were sparse and high-dimensional, resulting in many false positives (S. He et al., 2016). The event count matrices used in this study are without a doubt high-dimensional, which supposedly explains LogCluster's difficulties. The developers of Invariants Miner and LogCluster (Lou et al. (2010) and Lin et al. (2016), respectively) fail to address the limitations of their methods, and do not point to any types of data which might be unsuitable for the models, for example.

On the other hand, PCA and Isolation Forest achieve a solid accuracy in their anomaly detection. Especially when using log files parsed with AEL, both models were able to detect real anomalous sequences in the test file 174_3_7, regardless of the used training data. Moreover, their processing times are very low, even less than 30 seconds with AEL-parsed log files. The amount of false positives are quite high, but at an acceptable level, being three falsely classified sequences with both models, when using Time sequences. On the other hand, PCA detected only one of the real anomalous sequences, whereas Isolation Forest detected both. For these reasons, the final version of *sgologs* parses input logs with AEL, and uses PCA or Isolation Forest for the anomaly detection.

The nature of anomalies which are easy to detect was also clarified. Additional log entries are much simpler for the machine learning models to classify than the absence of normal log entries. For example, even though one or more normal sequences are missing in 174_3_18, there is nothing abnormal in the remaining sequences that would point to this anomaly. It would not be justified for the machine learning to classify a normal sequence as an anomaly if it is not followed by another normal sequence. This is why the real anomalous sequences in 174_3_7 were detected, and why the models did not find anything relevant in 174_3_18.

Additional error messages that do not usually appear in these log files are a good precedent of an anomaly. They are also the information developers most likely search for in the log files, with or without anomaly detection. It is promising that the models in *loglizer* are able to find these extra errors, resulting in *sgologs* being able to produce an output where the anomalies are ready for a convenient inspection.

9. Second Iteration

This section presents the development of the final artefact, *sgologs*, as well as the evaluation of the tool. Regarding the Design Science Research process model, the second iteration encompasses the same activities as the first iteration: design and development, demonstration, and evaluation. The process of a log file going through *sgologs* is illustrated in Figure 10, along with how the *logparser* and *loglizer* toolkits are used within the tool. Unlike *sgologs_alpha*, normal, unstructured log files are accepted as input, since a tool from *logparser* is incorporated in *sgologs*.

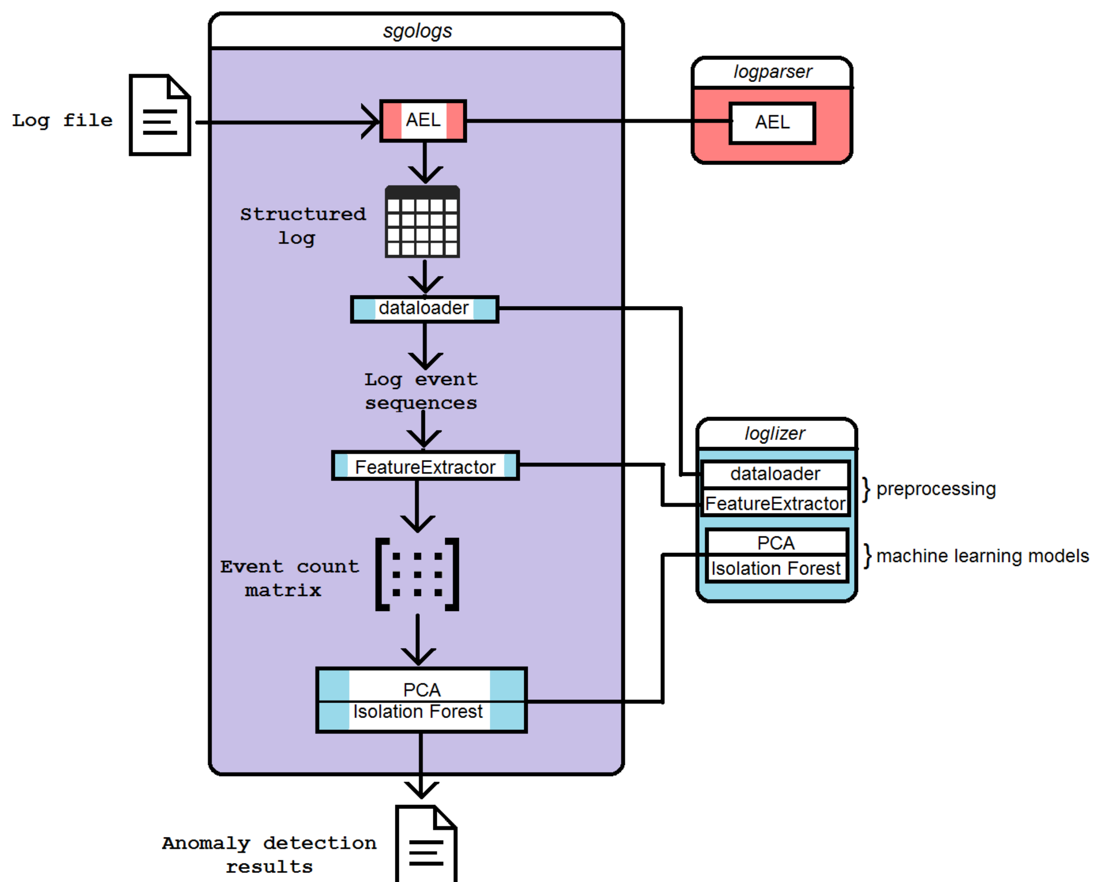

Figure 10. The anomaly detection process of *sgologs* with different parts of *logparser* and *loglizer*

Structured log files can also be fed to *sgologs*, in which case the log parsing phase is simply skipped. The differences between *sgologs_alpha* and *sgologs* are discussed in section 9.1. Then, section 9.2 presents the anomaly detection results with sequences determined with fixed windows, which were not used during the first iteration.

9.1 Changes to *sgologs_alpha*

The *logparser* tools, by default, append "_structured" to end of the log file name, and save the structured log file as a CSV file. Thus, structured log files are still accepted as input to *sgologs*, but if the name of the input file does not end with "_structured.csv", the log parser is used. Since AEL was found to be the best parser for these log files in section 7, and the best anomaly detection accuracies were achieved with AEL-parsed logs in section 8, it is used for log parsing in *sgologs*. Conveniently, the processing time of AEL for a single log file is only a fraction of a second, so the processing of *sgologs* is not considerably slower than that of *sgologs_alpha*.

During the first iteration, session windows based on PID values were found to be unsuitable. Likewise, the LogCluster model did not achieve an acceptable accuracy in its anomaly detection. Therefore these two features were dropped from *sgologs* altogether. The available machine learning models in *sgologs* thus are PCA and Isolation Forest, both of which were able to detect anomalies, as discussed in section 8.4. For determining event sequences, there are two options in *sgologs*: session windows based on Time values, and fixed windows.

Implementing the fixed windows was a fairly simple task. A new function, `slice_fixed`, was written into *loglizer*'s `dataloader` module. In the function the log events of the structured log are simply arranged to sequences based on their row number, and the size of each sequence is controlled with the `window_size` variable. Padding is added to the last sequence to ensure the length of each sequence is the exact same, i.e. the value in `window_size`. This way, the anomaly detection should not classify the last sequence as anomalous because it is shorter than all the other sequences.

The parameters of the machine learning models needed to be adjusted, as the parameters used with the Time sequences were unsuitable. For example, PCA's threshold for anomaly detection was originally set to 1400.0, but when fixed windows were used, it had to be lowered to 350.0. With fixed sequences and a threshold of 1400, PCA did not classify anything in the test files as anomalous, i.e. even the amount of false positives was zero. *Sgologs* automatically sets the appropriate values for the parameters, depending on the selected sequence determination style.

9.2 Evaluation of *sgologs*

Here, the results *sgologs* achieved with fixed sequences are described. Initially the fixed window size was set to 50, i.e. each event sequence contained 50 log events. Thus the test files were organized into sequences as follows: the file 174_3_7 has 56 sequences, 174_3_11 has 36 sequences, and 174_3_18 has 32. All training and test log files were parsed with AEL only. The results with PCA are presented in Table 12, and results with Isolation Forest are presented in Table 13.

Table 12. Anomaly detection results of PCA with fixed sequences (window size 50)

Test file (training data)	Real anomalous sequences	Detected anomalous sequences	True positives	False positives	False negatives
174_3_7 (A_170)	4	7	0	7	4
174_3_7 (B_178)	4	7	0	7	4
174_3_11 (A_170)	-	3	-	3	-
174_3_11 (B_178)	-	2	-	2	-
174_3_18 (A_170)	1	9	1	8	0
174_3_18 (B_178)	1	10	1	9	0

Table 13. Anomaly detection results of Isolation Forest with fixed sequences (window size 50)

Test file (training data)	Real anomalous sequences	Detected anomalous sequences	True positives	False positives	False negatives
174_3_7 (A_170)	4	8	0	8	4
174_3_7 (B_178)	4	7	0	7	4
174_3_11 (A_170)	-	7	-	7	-
174_3_11 (B_178)	-	5	-	5	-
174_3_18 (A_170)	1	7	0	7	1
174_3_18 (B_178)	1	6	0	6	1

Contrary to the results achieved with Time session windows during the first iteration (discussed in section 8), the anomaly detection accuracy with fixed windows was unsatisfactory. PCA and Isolation Forest both classified many sequences as anomalies, but failed to detect real anomalies. Therefore the amounts of false positives and false negatives were high. Surprisingly, PCA classified the last sequence in 174_3_18 as an anomaly, which could be considered a true positive. However, it is still unclear why PCA determined that the last sequence is anomalous, and it is possible that the correct classification occurred by chance. On average, PCA processed for 30 seconds and Isolation Forest for 36.5 seconds.

Figure 11 depicts the best scores PCA achieved with fixed window size 50 and the test files 174_3_7 and 174_3_18. Real anomalies in 174_3_7 were not detected, and thus all the scores are zero. The one and only anomaly in 174_3_18 was detected, resulting in maximum recall, but the detection was made at the expense of numerous false positives, resulting in a poor precision score.

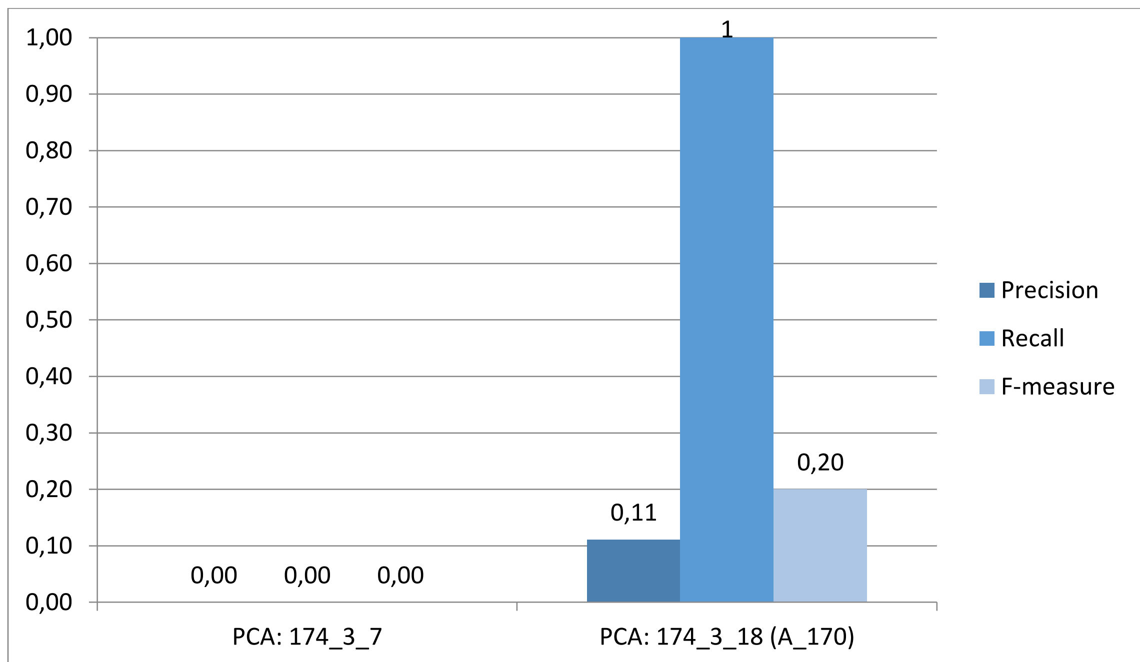

Figure 11. PCA's scores with fixed sequences (size 50) on files 174_3_7 and 174_3_18

Different sizes for the fixed windows were also tested. When the window size was set to 30, PCA correctly classified one of the four real anomalies in 174_3_7, but did not classify the last sequence in 174_3_18 as anomalous. The performance of Isolation Forest was not improved, compared to using a window size of 50 log events. Moreover, processing times of both models were increased to almost one minute, as the amount of sequences was higher.

With 100 as the fixed window size, the test file 174_3_7 has 28 sequences, 174_3_11 has 18, and 174_3_18 has 16. Increasing the window size had a positive effect. First, as the amount of event sequences was reduced, average processing times were much lower: 19.5 seconds with PCA and 19.9 with Isolation Forest. Second, PCA achieved a higher accuracy. The amount of detected real anomalies in 174_3_7 was doubled from one to two, and the amount of false positives was lower, compared to window size 30. Unfortunately, the true positives for Isolation Forest remained at zero. The results can still be considered better than when using sequences of 50 events, because the amount of false positives was lower. The results achieved with fixed windows of 100 log events are presented in Tables 14 and 15.

Table 14. Anomaly detection results of PCA with fixed sequences (window size 100)

Test file (training data)	**Real anomalous sequences**	**Detected anomalous sequences**	**True positives**	**False positives**	**False negatives**
174_3_7 (A_170)	4	6	2	4	2
174_3_7 (B_178)	4	7	2	5	2
174_3_11 (A_170)	-	5	-	5	-
174_3_11 (B_178)	-	1	-	1	-
174_3_18 (A_170)	1	7	1	6	0
174_3_18 (B_178)	1	7	1	6	0

Table 15. Anomaly detection results of Isolation Forest with fixed sequences (window size 100)

Test file (training data)	**Real anomalous sequences**	**Detected anomalous sequences**	**True positives**	**False positives**	**False negatives**
174_3_7 (A_170)	4	5	0	5	4
174_3_7 (B_178)	4	5	0	5	4
174_3_11 (A_170)	-	4	-	4	-
174_3_11 (B_178)	-	5	-	5	-
174_3_18 (A_170)	1	3	0	3	1
174_3_18 (B_178)	1	2	0	2	1

Figure 12 shows PCA's scores with size 100 fixed windows. Isolation Forest was excluded from the Figures because the amount of true positives was zero with all fixed windows.

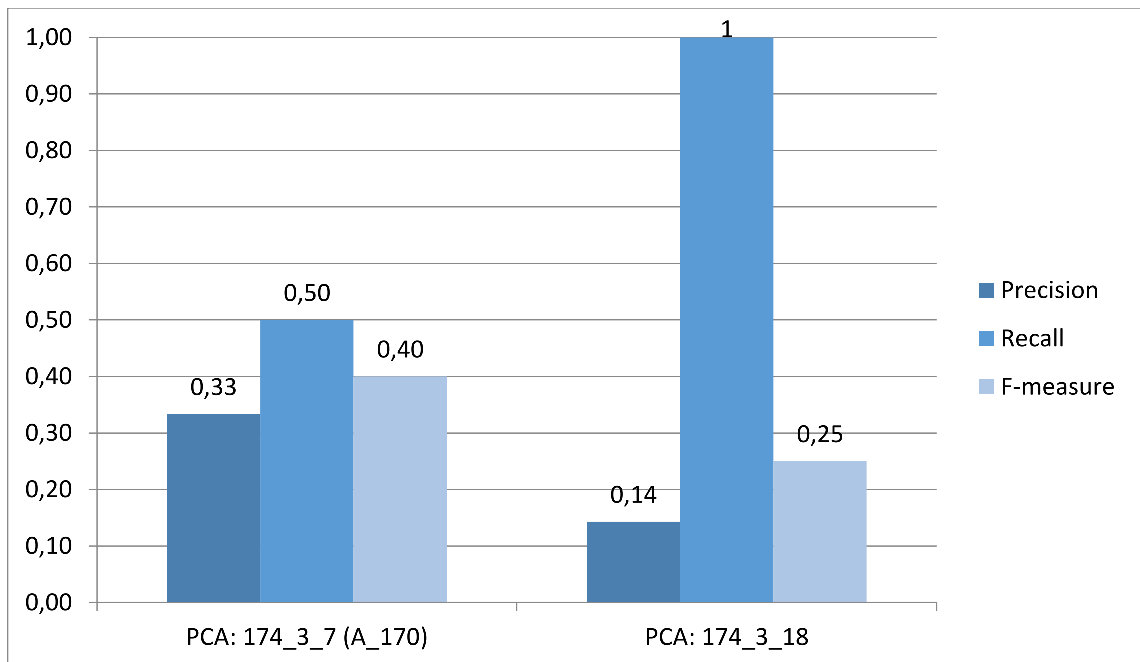

Figure 12. PCA's scores with fixed sequences (size 100) on files 174_3_7 and 174_3_18

Overall, PCA performs the best with fixed windows (size 100), even when compared to Time session windows, described in section 8. When analysing test file 174_3_7, the precision score was better, even though recall unfortunately remained at 0.5. Precision achieved with test file 174_3_18 is admittedly unimpressive.

PCA remains a valid anomaly detector with fixed windows, at least when the window size is set to 100. One disadvantage of such a large window size is that the output files, where the message contents of the anomalous sequences are written, can become quite long. For example, seven detected anomalies result in 700 log messages in the output file. Reducing the window size and simultaneously retaining the anomaly detection accuracy could be possible if sliding windows were used. After all, S. He et al. (2016) found that sliding windows are the best grouping technique. This investigation is left for future research. Nevertheless, when processing the file 174_3_7, the ratio of true positives, false positives, and false negatives is similar here as when using Time sequences. Moreover, PCA also managed to classify the last sequence in 174_3_18 as an anomaly, and the model's performance was stable against both training data sets.

The fact that Isolation Forest was unable to find anomalies when using fixed windows is disappointing. Different parameters for the model did not improve the situation. Isolation Forest is still a credible choice for anomaly detection when session windows based on Time values are used, as discussed in section 8. However, the model apparently struggles with sequences which are uniform in length. The researchers behind Isolation Forest state that the method performs well especially with large databases, where the number of instances exceed 100 000 (Liu et al., 2008). It is possible that a log file with only a few dozen event sequences is simply too small for Isolation Forest to work properly. Luckily fixed windows can still be used with PCA, but it would have been better for the artefact to always have two trustworthy machine learning models to choose from, regardless of the window type used for the sequence determination.

10. Discussion

This section discusses several considerations regarding this study, as well as its implications. The research questions specified in section 2 are revisited in 10.1. Other considerations can be found from section 10.2.

10.1 Answers to the research questions

Unfortunately, there are no straightforward answers to the research questions. As in most machine learning and anomaly detection studies (see e.g. Chandola et al., 2009), several assumptions for example about the data and the nature of the anomalies had to be made, and in another context the assumptions might be considerably different.

The first research question considers the validation of the anomaly detection results. As a reminder, the question was formulated as follows:

> *RQ1: Can the accuracy of the anomaly detection be effectively verified by the amount of detected errors?*

Error messages in log files are a relevant aspect. Work flow errors, for example, which manifest as error messages in system logs, are a typical anomaly and an interesting feature in troubleshooting and problem diagnosis tasks (Fu et al., 2009; Lou et al., 2010). Developers are interested in the error messages, and the machine learning should be too. It is relatively safe to assume that potential users of *sgologs* expect to see many error messages in the output. Therefore, the amount of detected error messages is a relevant factor in measuring the accuracy of the anomaly detection. For example, the additional error messages in the test file 174_3_7 clearly are an anomaly, and it is relevant to verify whether the machine learning recognised the errors.

However, the test files might have other anomalous behaviour as well. For example, some process may have completed unexpectedly late, or some "normal" log entries might be completely missing. Thus, it is completely plausible that some of the sequences classified as anomalies by the models, which were considered false positives, are in fact true positives. This is the danger of focusing only on errors in the validation of the results. Since it was unfeasible to manually classify all sequences in the test files, instead of only the most obvious anomalies, the possible non-error anomalies are not represented in the accuracy scores. In other words, the scores can be thought to only measure the accuracy of *error detection*, instead of more comprehensive anomaly detection.

In this context the focus on errors in validation of the results surely serves its purpose. Depending on the viewpoint, this focus could be a valid representation of accuracy. Nevertheless, an affirmative response to research question one should definitely be followed by "but only to some degree."

Research question two is about the accuracy the machine learning models are able to achieve. The question was formulated in section 2.2 as follows:

RQ2: How accurate is the anomaly detection with unsupervised machine learning and Linux log files?

A short answer is 'not very accurate.' The F-measure score only exceeded 0.5 when session windows based on Time values were used for sequence determination, and Isolation Forest was used as the machine learning model. Even the score in question, 0.57, is quite modest, as S. He et al. (2016), for example, measured scores as high as 0.98 with unsupervised methods. Moreover, their worst F-measure score, 0.5, is quite close to the best score measured in this study.

A major factor in the low F-measure scores is low precision scores, which were the result of several false positives. This finding is not surprising, as a high amount of false positives is characteristic to unsupervised learning (Landauer et al., 2018). This claim is confirmed in this study.

As discussed in previous sections, false negatives are considered more severe in this study than false positives. F-measure, also called F_1-measure, puts equal emphasis on precision and recall, i.e. on false positives and false negatives. However, an alternative calculation exists, where more weight is put on recall: the F_2-measure. By emphasizing recall, which is considered more critical in this study, the overall accuracy scores are improved significantly. Figure 13 shows the accuracy scores with AEL as the log parser, session windows based on Time values as log event sequences, and 174_3_7 as the test file.

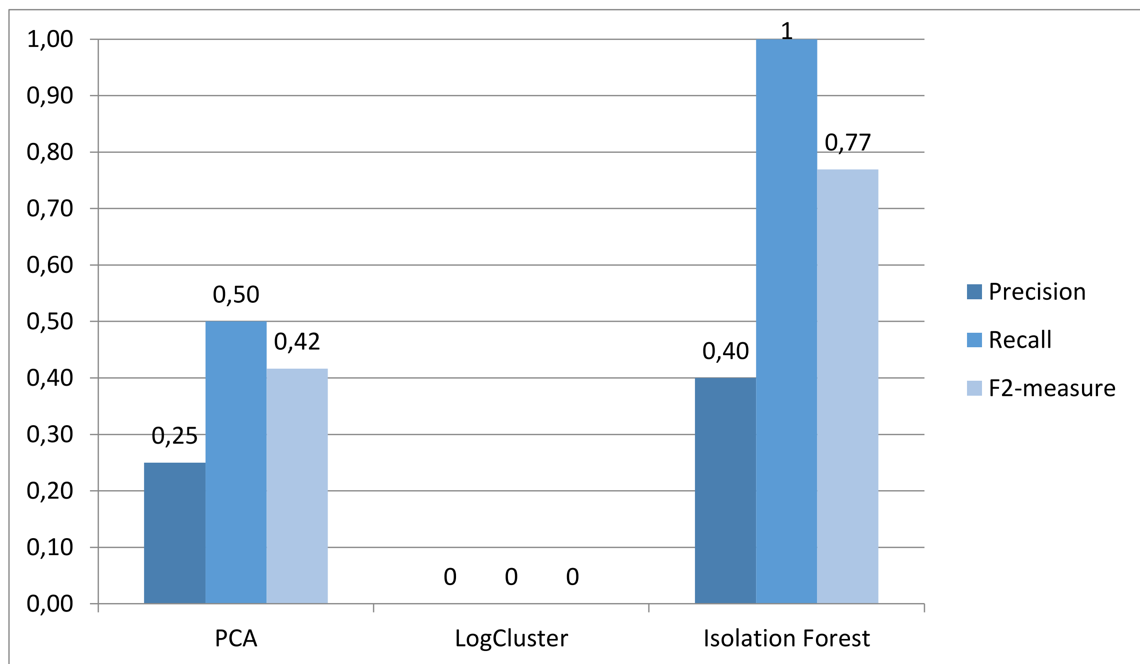

Figure 13. Accuracy scores of the models with Time sequences and file 174_3_7, using F_2-measure

Compared to Figure 9, where F-measure was used, the F_2-score is considerably higher. Especially with Isolation Forest, the 0.77 is a rather notable result. PCA did not get an equally dramatic score increase, as recall is still only 0.5. The situation is similar as with fixed sequences, as shown in Figure 14.

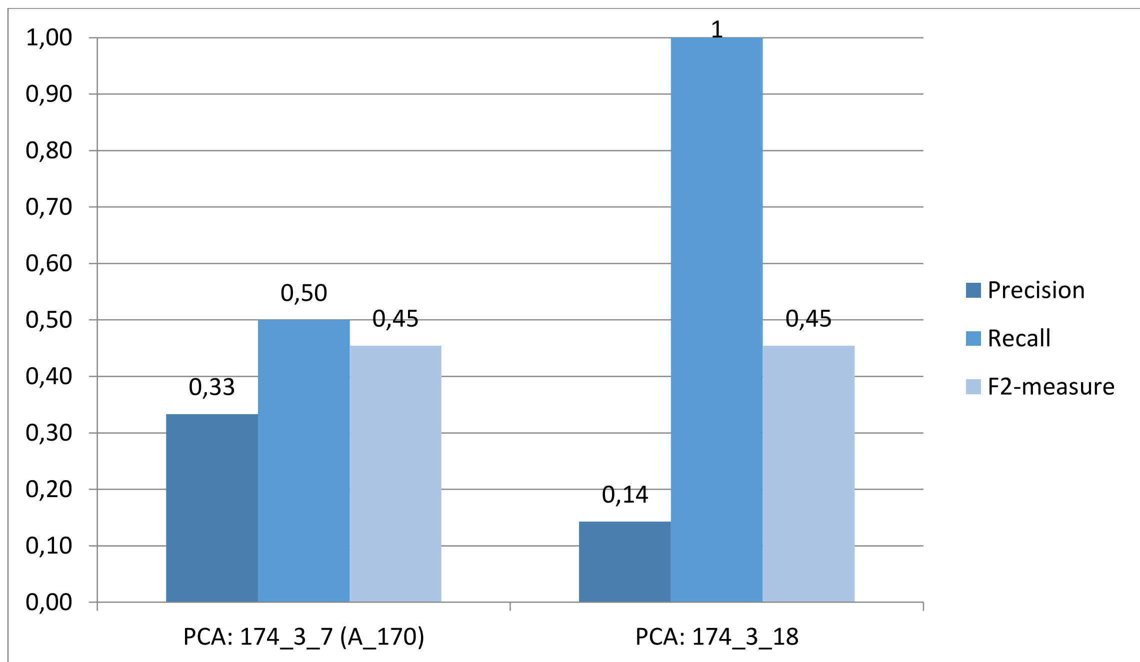

Figure 14. Accuracy scores of PCA with fixed sequences (size 100) and files 174_3_7 and 174_3_18, using F_2-measure

Even when emphasizing recall, PCA's score does not unfortunately exceed 0.5. This is understandable, since recall with test file 174_3_7 is only 0.5, but it is also notable that the F_2 score with 174_3_18 is 0.45, even though precision is only 0.14.

Again, the response to the research question is highly dependent on the view of the researcher. If minimizing false positives is considered as crucial as minimizing false negatives, i.e. emphasis on precision and recall is equal, the accuracy scores surely seem insufficient. The best F_1 score measured here is only 0.57, which is achieved only by one model, when using specific sequences and a certain test file. Moreover, the best F_1 score with the test file 174_3_18 is only 0.4.

However, if recall is emphasized more than precision, the accuracy scores seem much better. The best F_2 score of 0.77 is relatively acceptable, though it is still achieved by one model only, in quite specific circumstances. Moreover, even though the model in question, Isolation Forest, is clearly the best option for test file 174_3_7 when Time sequences are used, it is unable to detect that test file 174_3_18 is abnormally short, and is useless when fixed windows are used to determine the sequences.

Zhu et al. (2019) noted that Linux log files have complex structure and an abundance of event templates, which makes their accurate, automated parsing difficult. The same reasons most likely also affect the anomaly detection accuracy. In their study, S. He et al. (2016) found that LogCluster struggles with the event count matrices generated from BGL data, because the matrices are sparse and high-dimensional. As the amount of event templates in the Linux log files used in this study is in the hundreds, so are the lengths of the event count vectors. Thus, the event count matrices end up being very high-dimensional. This is quite clearly the reason why LogCluster is unusable (S. He et al., 2016), but it might negatively influence the other machine learning models as well.

An accuracy of e.g. 0.57 might be enough for the utility of *sgologs*, but the user should be aware of a few things. Firstly, sequence determination is an important consideration. For these log files, session windows with Time values seem to be the most serviceable

choice, as PCA and Isolation Forest were both able to detect anomalies in test file 174_3_7. Moreover, Isolation Forest is the best choice with Time sequences, as the measured F_1 score was 0.57 and F_2 was 0.77. However, if the user wishes to use fixed sequences, PCA is the only functional choice.

Secondly, the user should know what kinds of anomalies are simpler to detect. Additional log entries, such as error messages, which do not commonly occur in the log files, have a higher chance of being classifies as anomalous. With these things in mind, *sgologs* could be considered to be accurate enough, since at least some anomalies are detected and reported. The user is at least pointed in the right direction, although the accuracy scores are evidently modest.

10.2 Other considerations

No supervised learning. Only unsupervised machine learning was used in this study, since generating label files, which are required for supervised learning, requires considerable effort (Chandola et al., 2009). Even though supervised learning is rarely a practical possibility (S. He et al., 2016; Geijer & Andreasson, 2015), it most likely would have achieved better anomaly detection accuracy. Fortunately, unsupervised learning has been proven to be able to detect anomalies in previous work (Landauer et al., 2018; S. He et al., 2016), as well as in this study.

Log parsing. As *loglizer* requires structured logs as input, it is very fortunate that the open-sourced *logparser* toolkit is available. Manual log parsing would be, again, a very labour-intensive task (P. He et al., 2016). Log-based anomaly detection typically focuses on execution path anomalies via log keys, or log event templates, extracted from the log messages (Du et al., 2017). This is possible because each log key type typically corresponds to one log print statement in the source code, resulting in log key sequences representing the execution path of the software (Fu et al., 2009). However, it is also possible to consider other features in the log entries, such as timestamps and parameter values, in order to detect different kinds of anomalies, like DeepLog does (Du et al., 2017). In fact, an implementation of DeepLog, which relies on label data, is included in *loglizer*.

The performance of the *logparser* tools in this study was surprising. Even though Zhu et al. (2019) measured LenMa to achieve the best parsing accuracy with Linux log files, the tool provided disappointing results here. The AEL tool clearly performed the best, even though it only reached third place in Zhu et al.'s (2019) benchmarking. However, it should be noted that log parsing accuracy was not accurately measured here, as manually parsing a log file to represent the ground truth was beyond the scope of this study. Nevertheless, AEL seemed to be accurate in its parsing, and also improved the anomaly detection accuracy compared to log files parsed with SHISO. In addition, as another positive feature, the processing time for a single log file was only a fraction of a second.

Windowing techniques. It appears that log event sequences which retain the occurrence order of the log events in relation to each other are the most suitable for the machine learning models. Such sequences can be generated with fixed or sliding windows, and possibly with session windows, if the timestamps of the log entries are used as the session parameter. As most processes overlap in a typical Linux log, session windows based on other parameters, such as PID, cannot carry information about event order. Therefore the execution path is not reflected in the sequences as explicitly – a possible

reason why the models were unable to detect the correct anomalies in the PID sequences. On the contrary, S. He et al. (2016) argue that such session windows result in higher correlation between the log messages, which has a positive effect on the anomaly detection accuracy. However, this result was produced with HDFS logs, i.e. the used data was drastically different from the Linux log files used in this study.

Anomaly detection. It is unfortunate that InvariantsMiner, the tool measured to be the most accurate anomaly detector in S. He et al. (2016), had an unacceptably long processing time, as discussed in section 8. It is possible that the model could have reached a desirable performance with some optimizations, i.e. using different kinds of windowing techniques and log sequences, or making some changes in the implementation, for example. Unfortunately, this was not feasible within the time frame of this study. The same goes for LogCluster; however, it is possible that the Linux log files are simply too high-dimensional and effectively unsuitable for the model.

11. Conclusion

Log files are a relevant source of information, especially in troubleshooting and debugging tasks (Dunaev & Zaytsev, 2019). Due to the fact that log files tend to be complex and large, there is much demand for automated log file analysis (S. He et al., 2016). One common approach is to leverage machine learning to detect anomalies, which typically indicate faulty behaviour of the software (Fu et al., 2009). Many existing studies about log-based anomaly detection exist, and they have even resulted in the open-source toolkit *loglizer* (S. He et al., 2016). However, Linux log files have not been extensively used in existing research. This study attempts to fill this gap.

The design artefact of this study, *sgologs*, was developed around the open-source toolkits *logparser* and *loglizer*. Tools based on unsupervised learning were focused on, because the approach is much more convenient and applicable in practical settings (Geijer & Andreasson, 2015). Taking a Linux log file as input, *sgologs* parses the input file into structured data, and is able to relatively successfully detect anomalies in the log. Unfortunately, the accuracy of the anomaly detection depends considerably on a few factors. The method of arranging the log messages into sequences is an important consideration. In this study, session windows based on Time values in the log entries were found to enable the best accuracies. In addition, the best overall accuracy score was achieved with the Isolation Forest model. PCA achieved lower accuracy, but was able to correctly detect at least some of the real anomalies in the test files. Moreover, PCA was the only suitable machine learning model when fixed windows were used in sequence determination.

Linux log files seem to be challenging for automated processing. *Logparser* tools have been measured to achieve much better parsing accuracies with log files from other systems (Zhu et al., 2019). Nevertheless, the AEL parser seems to work sufficiently for these logs, and therefore it was included in *sgologs*. Moreover, the anomaly detection accuracy of the models from the *loglizer* toolkit was quite modest in this study. Two of the tested machine learning models, Invariants Miner and LogCluster, were completely incompatible with this data.

However, if the user of *sgologs* is aware of these factors, the tool can help in finding non-conventional entries in Linux logs. Especially if the anomaly is some additional material in the log file, *sgologs* has a high chance of detecting it, even though only unsupervised learning models, practical but not so accurate, are (currently) available. The user should also be prepared for noticeable amounts of false positives.

Sgologs was developed at a midsized ICT company, and it could be used as a part of the developers' everyday debugging tasks. The tool certainly has room for improvement, but integrating new features should be very straightforward, as it was during the development of the artefact. At the same time, there is room for numerous possibilities for future research in log-based anomaly detection with Linux logs.

11.1 Limitations

At numerous points in this study alternative methods could have been used, if they did not require work-intensive preparations. For instance, there is no sure way of verifying the accuracy of log parsing. This is because exact verification would have required comparison to a manually parsed log file, and creating such a file is too arduous to fit in the scope of this thesis. Likewise, creating label files for the test log files would have required excessive effort.

Label files would have enabled the usage of supervised learning in the anomaly detection, and aided in the validation of the anomaly detection results. S. He et al. (2016) measured supervised learning to be more accurate in log-based anomaly detection than unsupervised learning. Supervised learning would probably have been the more reliable method in this context as well. Comparing the achieved anomaly detection results to a label file would have enabled an exact and robust measurement of the accuracy, as in S. He et al. (2016). Generating a label file for each used test file would result in scrutinizing hundreds of different kinds of log sequences, classifying each as normal or anomalous.

Instead of the laborious and thorough labelling process, only the most obvious anomalies in the test log files were identified. The identification was based on the fact that the amount of error messages in some of the test files was unusual. The validation of the results was then based on these identifications, as discussed in section 2.1. It is possible that the machine learning correctly detected anomalies which were not manually identified as anomalies, i.e. sequences which do not contain unexpected errors. Thus the accuracy could, in theory, be better than what is presented here.

Moreover, the selection of the test data was unconventional for a machine learning study. Typically, the test files are selected at random, after which possible anomalies in them are identified. In this study, the log files were included in the test data if they contained an abnormal amount of error messages, i.e. anomalies. Thus, test file selection can be considered biased. However, as the amount of test files is only three, statistically significant conclusions are not possible anyway. Nevertheless, even with the unconventional test file selection, investigating the potential of the *loglizer* tools in this context was possible.

11.2 Future research

If at least some developers started using *sgologs* as a part of their routine log inspection tasks, an opportunity for a clear follow-up study would emerge. The developers could be surveyed and interviewed about the benefits and shortcomings of the anomaly detection tool. This way, the actual real-life utility of *sgologs* could be investigated. At this stage, however, only educated guesses about the adequacy of the tool can be made. For example, a survey and interview-based study would probably reveal whether a F_1 score of 0.57 is at all sufficient in practice.

However, based on this study, it is already apparent that some parts of the artefact could use some polishing. For instance, the fact that the machine learning models need to be trained each time the tool is run is a considerable issue. Changing the implementation so that the models somehow store the parameters they learned from the training data is plausible and possibly simple. With such an implementation, much larger training datasets could be used, since the processing time of the training phase is no longer a

relevant consideration. Increased training data size would supposedly enable better learning, which could in turn increase the anomaly detection accuracies.

Another implementation currently lacking in *sgologs* is the option to use sliding windows to determine the log sequences. The technical implementation would be a fairly simple task, but the positive effects to the detection results could be remarkable. After all, the window sequence type is not a trivial matter, as discussed in earlier sections. Another interesting possibility is detecting anomalies at the level of individual log messages, i.e. not using sequences at all. This would simplify the output greatly, as only single log rows would be classified as anomalous. If the *loglizer* tools could be reshaped to work like this, the utility of the output would most likely increase.

Going through the trouble of generating label files for the Linux logs, thus allowing the usage of supervised learning models, could be beneficial. The *loglizer* toolkit already contains three supervised models (LR, Decision Tree, and SVM), so model implementation does not require great effort. Label files, although laborious to make, would most likely be worth the effort, if supervised learning really achieves higher accuracies than unsupervised learning, as measured in previous work.

Moreover, there are also unsupervised options which have not been investigated in this study. At least partial implementations, which rely on label files, of DeepLog, Local Outlier Factor (LOF), one-class SVM, and AutoEncoder, are already included in *loglizer*. Especially DeepLog, based on deep learning, seems promising. If label files are off the table, changing the implementation of the models to not require labelled training data should not be a demanding endeavour. It would be fruitful to investigate the other unsupervised models. It is well possible that the performance of the models would be comparable to the models tested in this study, or even exceed the accuracy scores measured here.

References

Bovenzi, A., Brancati, F., Russo, S., & Bondavalli, A. (2015). An OS-level framework for anomaly detection in complex software systems. *IEEE Transactions on Dependable and Secure Computing, 12*(3), 366-372. doi:10.1109/TDSC.2014.2334305

Chandola, V., Banerjee, A., & Kumar, V. (2009). Anomaly detection: A survey. *ACM Computing Surveys, 41*(3), 1-58. doi:10.1145/1541880.1541882

Cheng, X., & Wang, R. (2014). Communication network anomaly detection based on log file analysis. Paper presented at the 9th International Conference on Rough Sets and Knowledge Technology, Shanghai, China. doi:10.1007/978-3-319-11740-9_23

Du, M., Li, F., Zheng, G. & Srikumar, V. (2017). DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. Proceedings from CCS '17: *The 2017 ACM SIGSAC Conference on Computer and Communications Security.* Dallas, TX, USA.

Dunaev, M. & Zaytsev, K. (2019). Logs analysis to search for anomalies in the functioning of large technology platforms. *Journal of Theoretical and Applied Information Technology, 97*(11), 3123-3135.

Fu, Q., Lou, J.-G., Wang, Y. & Li, J. (2009). Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. Paper presented at the 2009 Ninth IEEE International Conference on Data Mining, Miami, FL, USA. doi:10.1109/ICDM.2009.60

Geijer, C. & Andreasson, J. (2015). Log-Based Anomaly Detection for System Surveillance (Master's thesis). Retrieved from Chalmers Publication Library. (219089)

He, P., Zhu, J., He, S., Li, J. & Lyu, M.R. (2016). An Evaluation Study on Log Parsing and Its Use in Log Mining. Paper presented at the 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Toulouse, France. doi:10.1109/DSN.2016.66

He, S., Zhu, J., He, P. & Lyu, M.R. (2016). Experience Report: System Log Analysis for Anomaly Detection. Paper presented at the 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), Ottawa, ON, Canada. doi:10.1109/ISSRE.2016.21

Henrique, B. M., Sobreiro, V. A. & Kimura, H. (2019). Literature review: Machine learning techniques applied to financial market prediction. *Expert Systems with Applications, 124*, 226-251.

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly, 28*(1), 75-105. doi:10.2307/25148625.

Jiang, Z., Hassan, A.E., Flora, P. & Hamann, G. (2008). Abstracting Execution Logs to Execution Events for Enterprise Applications (Short Paper). Proceedings from QSIC '08: *The 8th International Conference on Quality Software.* Oxford, UK. doi:10.1109/QSIC.2008.50

Khan, A., Baharudin, B., Lee, L.H. & Khan, K. (2010). A Review of Machine Learning Algorithms for Text-Documents Classification. *Journal of Advances in Information Technology, 1*(1), 4-65.

Kim, J., Minsik, P., Kim, H., Cho, S. & Kang, P. (2019). Insider Threat Detection Based on User Behavior Modeling and Anomaly Detection Algorithms. *Applied Sciences, 9*(19), 1-21. doi:10.3390/app9194018.

Landauer, M., Wurzenberger, M., Skopik, F., Settani, G. & Filzmoser, P. (2018). Dynamic log file analysis: An unsupervised cluster evolution approach for anomaly detection. *Computers & Security, 79*, 94-116.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature, 521*, 436-444. doi:10.1038/nature14539

Leal-Aulenbacher, I. & Andrews, J.H. (2013). Generating C++ log file analyzers. *WSEAS Transactions on Information Science and Applications, 10*(10), 313-324.

Lin, Q., Zhang, H., Lou, J., Zhang, Y. & Chen, X. (2016). Log clustering based problem identification for online service systems. Proceedings from ICSE '16: *The 38th International Conference on Software Engineering Companion.* Austin, TX, USA. doi:10.1145/2889160.2889232

Liu, F., Ting, K. M. & Zhou, Z. (2008). Isolation Forest. Paper presented at the 2008 8th IEEE International Conference on Data Mining, Pisa, Italy. doi:10.1109/ICDM.2008.17

Logpai. (2018, June 6). *AEL – Abstracting Execution Logs [README file]*. Retrieved March 2, 2020, from https://github.com/logpai/logparser/tree/master/logparser/AEL

Logpai. (2020, January 6). *Loglizer [README file]*. Retrieved March 5, 2020, from https://github.com/logpai/loglizer/blob/master/README.md

Lou, J.-G., Fu, Q., Yang, S., Xu, Y. & Li, J. (2010). Mining Invariants from Console Logs for System Problem Detection. Proceedings from USENIX ATC '10: *The 2010 USENIX Annual Technical Conference.* Boston, MA, USA.

Malhotra, R. (2015). A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing Journal, 27,* 504-518. doi:10.1016/j.asoc.2014.11.023

Mavridis, I. & Karatza, H. (2017). Performance evaluation of cloud-based log file analysis with Apache Hadoop and Apache Spark. *Journal of Systems and Software, 125*, 133-151. doi:10.1016/j.jss.2016.11.037

Mizutani, M. (2013). Incremental Mining of System Log Format. Paper presented at the 2013 IEEE International Conference on Services Computing, Santa Clara, CA, USA.

Mäkinen, M. (2017). *Deep Learning for Anomaly Detection in Linux System Log* [Master's thesis, Aalto University]. Aaltodoc. Retrieved from http://urn.fi/URN:NBN:fi:aalto-201906234158

Peffers, K., Tuunanen, T., Rothenberger, M. A. & Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems, 24*(3), 45-77.

Pietikäinen, M. & Silvén, O. (2019). *Tekoälyn haasteet: koneoppimisesta ja konenäöstä tunnetekoälyyn.* Retrieved January 8, 2020, from http://jultika.oulu.fi/Record/isbn978-952-62-2482-4

Shima, K. (2015). Length Matters: Clustering System Log Messages using Length of Words. Proceedings from CNSM '15: *The 11th International Conference on Network and Service Management.* Barcelona, Spain.

Systemd. (2020, January 11). Retrieved February 20, 2020, from https://wiki.archlinux.org/index.php/Systemd

Tsai, C.-F., Hsu, Y.-F., Lin, C.-Y. & Lin, W.-Y. (2009). Intrusion detection by machine learning: A review. *Expert Systems with Applications, 36*(10), 11994-12000.

Voyant, C., Notton, G., Kalogirou, S., Nivet, M.-L., Paoli, C., Motte, F. & Fouilloy, A. (2017). Machine learning methods for solar radiation forecasting: A review. *Renewable Energy, 105,* 569-582.

Xu, W., Huang, L., Fox, A., Patterson, D. & Jordan, M. (2009). Detecting large-scale system problems by mining console logs. Proceedings from SOSP '09: *The ACM SIGOPS 22nd symposium on Operating Systems principles.* Big Sky, MT, USA. doi:10.1145/1629575.1629587

Zhu, J., He, S., Liu, J., He, P., Xie, Q., Zheng, Z. & Lyu, M.R. (2019). Tools and Benchmarks for Automated Log Parsing. Proceedings from ICSE-SEIP '19: *The 41st International Conference on Software Engineering: Software Engineering in Practice*. Montreal, QC, Canada.