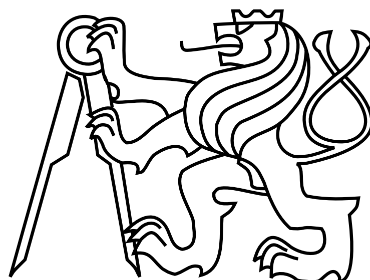
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science



Master's Thesis

Anomaly detection in complex software architectures

Bc. Ondřej Borovec

Supervisor: Ing. Martin Svatoš

Study Programme: Open Informatics

Field of Study: Data Science

May 24, 2019

Aknowledgements

I wish to thank my supervisor for his assistance and guidance. I would also like to thank my family and girlfriend for their support.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on May 24, 2019 ....

Motivation

During my studies I had an internship and later a regular job as a DevOps Engineer. The team, I was part of, is responsible for running a big cluster of NoSQL storage with surrounding infrastructure in multiple separated zones of the world as an internal logging platform. You can definitely image, we were facing some problems nearly every day and we were trying to figure out how to identify such problems and their the root causes. It was never easy, because during the down-times of our service we were under a big pressure going though log files and running several analysis tools since nobody in the company could monitor their own services.

At the time we would give nearly everything to have a reliable solution which would warn us in advance so we had more time to react. But we were not satisfied with any existing solution around. Since we were dealing with structured data and time series, my main field of study - Applied machine learning comes really handy.

Keeping a cloud application stable, healthy and alive, proved to be a heroic task and this is one of the main reasons, we choose to work to automate that procedure and to make life of other operators easier.

Ondrej Borovec

Abstract

Anomaly detection is a crucial aspect of software architecture maintenance and building a stable system. With early problem detection operators can react quickly to reduce potential downtime risk resulting in data and money saving, therefore a reliable real-time anomaly detection system is highly desired. Unfortunately, currently used monitoring techniques are lacking behind fast-growing industry applications and scale of used architectures.

This thesis aims at solving a problem of a renown company to design a new end-to-end solution for anomaly detection. We reviewed and discuss the best practices of designing such monitoring system, what anomaly detection techniques can be used, what metrics and features to collect and how to represent them. Collected logs and metrics by our system were preprocessed and released as a research dataset together with this work. The dataset records several days of anonymized runtime behaviour of 2 architectures with expert annotations of anomalous behaviour based on expert experience.

We identified potential weaknesses of current state-of-the-art models and propose a modification called Timed workflow inference to address this issue. We also designed and implemented a new graph-based model - Timed graph workflow - to generalize some strict rules of other solutions. Our models were experimentally evaluated with other state-of-the-art anomaly detection models using our dataset. The proposed solutions proved to be competitive and in several aspects even better.

Abstrakt

Detekce problémů a anomálií hraje důležitou roli při správě a tvoření komplexních softwarových řešeních. Brzká detekce potenciálního problémů pomáhá správcům takových systémů rychle reagovat a v důsledku snižovat riziko odstávky služby a ztráty peněz. Naneštěstí současná řešení pro monitoring zaostávají za rychle rostoucím I.T. průmyslem a velikostí samotných softwarových řešeních.

Tato práce je zaměřena na řešení problémů známé společnosti a má za úkol navrhnout nový kompletní řešení pro detekci anomálií. V rámci výzkumu této problematiky jsme se zabývali doporučenými řešeními monitorovacích systémů, jaké techniky detekce anomálií mohou být použity a které vlastnosti a příznaky architektur mají být sbírány a následně zpracovány. Námi navržený systém zaznamenával každodenní chování dvou různých architektur a tyto data jsou publikovány společně s touto prací jako vědecký dataset s anotacemi vytvořenými experty na dané architektury.

Identifikovali jsme potencionální slabiny uznávaných nejmodernějších metod a navrhli modifikaci jedné z nich na řešení tohoto problému. Také jsme navrhli a implementovali nový model založený na grafových strukturách sloužící jako generalizace současných řešení. Naše modely byly experimentálně vyhodnoceny v porovnání se zmíněnými uznávanými algoritmy na námi vytvořeném datasetu. Naše řešení se prokázalo být stejné kvality a v některých vlastnostech dokonce lepší.

Contents

1 Introduction **1**

2 State of the art **7**

2.1 Software logging and reliability . . . 8

2.1.1 Logging techniques . . . 9

2.1.2 Machine monitoring . . . 11

2.1.3 Problem detection practise . . . 13

2.2 Anomaly detection algorithms . . . 15

2.2.1 Supervised anomaly classification . . . 17

2.2.1.1 Support vector machine . . . 17

2.2.1.2 Decision trees . . . 19

2.2.1.3 K-Nearest Neighbors . . . 20

2.2.1.4 Supervised model comparison . . . 20

2.2.2 Unsupervised anomaly detection . . . 20

2.2.2.1 Log clustering . . . 21

2.2.2.2 PCA . . . 21

2.2.2.3 Invariant mining . . . 22

2.2.3 Semi-supervised anomaly detection . . . 23

2.2.3.1 One-class support vector machine . . . 24

2.2.3.2 Workflow inference . . . 25

2.2.3.3 Long short-term memory . . . 26

2.2.4 Comparison of anomaly detection models . . . 27

2.2.5 Time series analysis . . . 28

2.3 Anomaly detection evaluation . . . 29

3 Problem definition **33**

4 Dataset **35**
4.1 Existing datasets 35
4.2 ELKR dataset 36
4.2.1 Service architecture 37
4.2.2 Dataset structure 38
4.2.3 Recorded problems 41

5 Proposed solution **43**
5.1 Timed workflow inference 45
5.2 Timed graph workflow (TGW) 46
5.3 Timed hierarchical graph workflow 49

6 Experiments **51**

7 Conclusion **55**

Bibliography **57**

List of Figures

2.1 Log and metric collection schema. 14
2.2 Example decision tree anomaly detection 19
2.3 Supervised method comparison of f-measure, precision and recall 21
2.4 Simple process diagram. Dashed connection stands for potential fatal exceptions. 23
2.5 Sample automaton to illustrate workflow inference anomaly detection process. 25
2.6 Visualization of LSTM used in DeepLog. Image used from [19]. 27

4.1 Illustration of a logging platform in one environment. 38
4.2 Visualization of dataset 1 training phase 41
4.3 Visualization of dataset 1 42
4.4 Visualization of dataset 2 42

5.1 Workflow inference model automata for proposed execution 44
5.2 Illustration of hierarchy tree for logged records 49

6.1 Precision, recall and f-score comparison of studied models on architecture 1 of ELKR dataset 53
6.2 Precision, recall and f-score comparison of studied models on architecture 2 of ELKR dataset 54

List of Tables

2.1 Example of log transformation to log key format 10
2.2 Overview of log parsing frameworks 12
2.3 Log key sliding count matrix example 16
2.4 Anomaly detection algorithm overview 28
2.5 Machine learning algorithms used for anomaly detection with their efficiency. 28
2.6 Confusion matrix for anomaly detection 30

3.1 Problem confusion matrix 34

4.1 Attributes of architectures documented in datasets 38
4.2 Dataset attributes 40

5.1 Logged record sequence breaking standard Workflow inference algorithm . . . 45

6.1 Table of how many continuous anomaly behaviour were detected 53

Chapter 1

Introduction

The complexity of software architecture is growing with rapid speed and we cannot expect the incidence of failures or potential problems to decrease at the same rate. Therefore, modern systems are relying heavily on anomaly detection principles which can help with problem detection and in the best case even with their prediction for the potential design of self-healing systems such as the one described in [16].

Anomalies, often referred to, in other publications as outliers, exceptions or aberrations, may continually occur everywhere around us, whether we are able to distinguish them from standard behaviour depends only on our perspective and knowledge. It means, an event by itself is never an anomaly, to classify it we always need to know the context in which it occurs and, preferably, even some relevant history. To be more general, Cambridge dictionary defines an anomaly as follows[1]:

A person or thing that is different from what is usual, or not in agreement with something else and therefore not satisfactory.

But a more suitable for definition for us could be this one from an anomaly detection survey of 2009 [10]:

Anomalies are patterns in data that do not conform to a well-defined notion of normal behavior.

It means, an anomaly may be a number of a time series which does not belong there or for which the probability of observing is too low, according to expected rules or statistics. It can be a state of a state space which has previously been classified as an anomaly, has never been seen before or does not make a sense in current execution. It can be a sequence of actions which are not supposed to follow each other and there exist many other examples of patterns which do not fit to its context.

[1] http://dictionary.cambridge.org/dictionary/english/anomaly

A nice general example of an anomaly would be an e-shop where every transaction is logged and a time line of transaction per day is drawn. Such line will have, most probably, a week pattern with an increasing trend and slight fluctuation. You would be able to predict an amount of transaction with a certain probability level for most of days in year. But for instance, during a Black sale Friday the amount of transactions will go up. If a third party witnessed such a time line without context, they could expected that something went wrong (in a positive way), but an experienced business manager would know, it is normal and people are simply buying more goods.

The monitoring and problematic of anomaly detection in different forms is studied across multiple fields and domains. In statistics, this problem has been studied for long time with its roots in the 19$^{\text{th}}$ century - a historical paper [20] from 1887. There are many studies and implementations which deal with some form of anomaly or outlier detection to date. Let us highlight some to demonstrate possible implementations: [52] introduces a health-care-focused alerting system for wearable devices, [49] detects network traffic anomalies to discovers security threats, [7] studies credit card fraud detection and there are many other papers. The one thing all implementations have in common is that every field demands a specific approach and specific background knowledge. In other words, there is no universal algorithm or a solution which would be applicable to all problems. In general, a deep knowledge of data and its comprehension is the main and biggest key to finding the best anomaly detection approach.

Most of current cloud IT solutions and architectures are, like walls are built from bricks, built from multiple existing sub-components or applications - nobody develops from scratch, since it is easier and cheaper to use well-working sub-solutions to achieve new additional value. This means that, mandatory attributes such as stability and quality of new solutions do not only depend on the implementation of the system itself, but also on the implementation and usage of the sub-solutions used, hardware and network constraints. Complex software architectures are usually designed for 24x7 service, often for millions of users, thus the current standards of cloud services, for example, availability and reliability are crucial.

Before we go any further let us define our terms *complex software architecture*:

Definition 1.1. *A Complex software architecture can be a service or an application which is built on and is using multiple software instances as sub-solutions, which means there are communications and dependencies between these software instances, necessary to fulfill the desired functionality for the architecture and which spreads across multiple machines.*

Most sub-solutions often have their own self-healing processes, if they go to a simple error states, it is possible to recover without a big impact on their overall quality, but in high demand environment, you cannot only rely on that. Companies have many professionals (also called operators) who are responsible for quality of their services and who must deal

with any problem which occurs. From a personal experience, it may take quite some time to dig into all log files and find out what caused a problem. Even though a solution to a problem may be simple and solved rather quickly, every downtime is expensive as, for instance, British Airlines experienced from the end of May of 2017[2]. The best practice is to avoid totally such situations or at least be warned in advance, so you have a chance to prevent it.

We would like to highlight a real work experience with a message broker and a consumer storing data to a NoSQL database. We could restart each of the broken nodes and connection would renew automatically, but when there is a network drop, connection between these two component gets corrupted and data stop flowing. In that case, the consumer still can ping the broker and has a valid session thus it does not try to renew the connection, even though the broker already dropped it. If that happens, new messages are still being produced but not processed, the manual restart of all consumers is needed.

As it has already been mentioned, the best way to understand a software related problem is analysis of related log files. Those automatically generated text files contain messages designed by developers recording software states, action and status descriptions. And because the people who created the source code knows the most about potential problems, logs are the most valuable source of information. Unfortunately, it may also be tricky, since those messages are created by humans, other people do not have to understand them as they were intended to.

Every solid program and consequently every software architecture should have a deterministic behaviour. This means, there should always be a way to classify every potential state therefore it should be possible to assign a probability of potential problems. It means, software logs are perfect candidates for machine learning methods. Since the hardest analytic work would be designated to computers, operators do not have to spend their time with visiting each server and performing hand analysis of each log file to identify a root cause of an anomaly state they are facing. For the purpose of this work let us define *a potential anomaly state* of a software architecture:

Definition 1.2. *The potential anomaly state of a software architecture can be each state which on a pre-set probabilistic level leads to a situation which endangers functionally of the software architecture.*

Keeping in mind the e-shop example we used before, there are not only negation anomalies a system can experience. In software systems, it is possible to see a positive anomaly such as if resources of a virtual machine are increased and therefore percentage of CPU usage goes down. This kind of information is, in general, less useful than negative anomalies, which can lead to potential problems. Another problem can be the false-positive ratio, since

[2]https://www.cbronline.com/enterprise-it/ba-outage-understanding-true-cost-downtime/

warnings about potential anomaly states cannot be too frequent otherwise it may overwhelm everybody who follows them.

This thesis focuses on complex software architectures and runtime anomalous behaviour detection. To be more specific, we would like to address a need of an logging team of a company, which has to maintain hundreds of servers and deal with software infrastructure level problem, but did not find any feasible solution, so we were asked to find a solution to this problem.

Ultimately, in this thesis we would like to explore and summarize current approaches and designs for detection of anomalies in computer architecture behaviour, as well as suggest and test our own framework for a complex software architecture. There are many solutions around and even many good competitive commercial products, but still the results can be improved and there is a lot of research about this topic, so we decided to join the stream and target the real world anomaly detection problem.

In chapter 2, we summarize state of the art of console logs based anomaly detection as well as state of the art current log techniques and log message analysis and processing. We would like to give a reader at least a brief description of the algorithms and methods we are relying on with our implementation as well as to show what kind of performance could be expected from them. We are also in touch with leading log analysis and anomaly prediction companies and have tried to use their solution on existing datasets.

In the following Chapter 3 we summarize problematic we would like to address and in Chapter 4, we describe a new dataset, which was created to most precisely represent software architecture and problem we are aiming to solve. This new dataset was created in cooperation with a logging team of renounced international and it represents their internal logging platform solution which consists of multiple each-to-other-dependant applications on hundreds of servers. You can also find here some statistics about their records and dependencies. This dataset is published as a part of this thesis and given to scientific community for further usage and experiments.

To fulfill the main goal, we also suggest how an ideal solution could work, starting with how related data may be collected, stored and represent. But the main change to the current solutions is in designing and developing of new machine learning approach for unsupervised and semi-supervised learning using workflow analysis and graph-based models which are then evaluated on our new dataset along with other state of the art solutions. All proposed solutions are described in Chapter 5 and experiments and evaluation is located in chapter 6.

This work was influenced and is a result of cooperation of multiple research institutions and successful companies. The main consultations were made with researches at the Czech Technical University in Prague, the National School of Computer Science and Applied Mathematics of Grenoble and University of Massachusetts Lowell. We also consult results

and techniques with experts from SAP Concur[3] and Blindspot[4]. All participants brought valuable additional value in form of experience, support and innovative ideas and there is a big thanks to all of them.

All code and dataset can be found in Github repositorywear-ab[5].

[3] https://www.concur.com/
[4] http://blindspot-solutions.com/
[5] https://github.com/OBorovec/AnomalyDetectionAndPrediction

Chapter 2

State of the art

General anomaly detection technique has a wide range of important applications and the concept is applicable with just slight adaptations to nearly every field of human interest. There are hundreds of papers which are proving how useful it is to detect and know about anomalies in various area, but its efficiency varies cross all of them. It is intuitive, anomaly detection algorithms are more effective and successful in areas with strict rules, such as deterministic computer programs and software.

Even with the assumption of deterministic space of applications and software, anomaly detection in complex software architectures is not a trivial task. Generally speaking, really good domain knowledge of a target system is required to be able to correctly understand produced logs, time series and other metrics. This is the main reason why there is no 100% efficient general approach or tool so far.

To maintain a software architecture it is necessary to only detect a problem, but it is also interesting to know and understand the problem and reasoning behind. We all know the famous phrase from the series IT crows: "*Have you tried turning it off and on again?*" Most of the members of IT crowds will agree that it is most often the best and easiest way to fix anything, but if you do not know why it happened, it may cause some problems again. So, we will try to always discuss the potential of reasoning for every method listed below. Another important feature of autonomous anomaly detection framework is ability of continuous learning as discussed in [2] to reflect nature changes in underlying model statistics . Since we can expect, operators are able confirm an anomalous state we will also discuss how a method can use this new information.

In this chapter we will cover anomaly detection principles along with interesting methodologies related to logging, log collection architecture as well as open source frameworks and commercial products for IT architecture analysis. Most of the content of this chapter is not vital for understanding the technicalities of our contribution. It servers mostly as an overview to anybody who does not have a deep knowledge of general anomaly detection as

well as who is interested in current results of software anomaly detection. If you desire, you may skip this chapter and continue to Chapter 4 or Chapter 5 which are more related to our work.

2.1 Software logging and reliability

There are many different types of software, starting with a single stand-alone desktop application and reaching the sky with cloud applications connecting all kinds of wearable devices. All the software differ in features, target usage and scale, but we can still identify some features which are common to all of them. The first one we would like to highlight and have as ground truth to our work is that every application and architecture is deterministic and the only indeterminism is brought by an internet unreliability, hardware failure, system termination signal or others belonging to the same category.

This is an important statement, because during our discussions with many professionals in this field there was frequently asked this question: "Do we really need a research and machine learning tool for program logs since all programs are supposed to be deterministic?" We have to agree, this is a good question, but every application is be deterministic under the condition we know all potential aspects which can affect its run. And there is no chance we can stream such collection of information about every application with current possibilities.

This requirement becomes even more unreachable considering cloud applications which are the main scope of this work. For better understanding, let us list some determining features and best practice for cloud applications which are essential for our further statements.

- An architecture for a cloud application consists from micro services - it means there are multiple teams and projects which all are developed independently and individually are delaying only with small problems. The problem is the lack of deep cooperation between such micro services, so if one has a problem other does not have to know about it which can cause consequent problems.
- Such architecture has to be able to scale - it runs on multiple machines in different countries. This is mostly related to already mention problem of potential unreality of the internet. You can argue with potentially better protocols, but there is currently no way to ensure full steadiness.
- In the modern age, many teams and companies are abandoning standard version releasing in favour of continuous deployment, blue-green deployment, canary releasing and AB testing. It means, everybody can witness different states and behaviour cross same micro services.[1]

[1] http://blog.christianposta.com/deploy/blue-green-deployments-a-b-testing-and-canary-releases/

2.1.1 Logging techniques

Since the dawn of software development there was a need for recording of program behaviour, variable monitoring and reporting. Without it there is just a really narrow opportunity for anybody to be able to debug a running application or identify a potential problem. On the other hand, logging has been part of application development since nearly the beginning. Logging technique best practice developed significantly over the time from simple printing functions to complex libraries and frameworks like [22, 29].

To be able to image, what information and in what format can be expected in application console logs, lets us highlight some most important parts of logging best practise:

1. An application should be logging about itself as much as possible with each message containing a timestamp of related event along with a standard severity level.
2. Users should be able to customize logging possibilities of an application they are running.
3. Preferably use a standard logging library like Log4j, Log4net, boost.log or python logging instead of writing your own. Preferably, a framework which has a flexible output option and standardize formatting.
4. Every log message should be in format of a pattern string with a static and dynamic parts or a standard structure format as yaml or json with a predefined field mapping. The static part of each log message is a free-format string describing a run-time event or a property and the dynamic part is event of moment specific information.
5. Any action of your logger cannot block application itself for any reason such as inability of creating of a log file or dependent process for printing a message.

Unfortunately, even the best practice rules do not ensure the right logging form and content since there are not strict guidelines. We can ask ourselves three main questions related to logging topic: "What to log?", "Where to log?" and "How to log?". Solving these question have a significant effect on the value of provided information, software performance or disk I/O bandwidth. So, let us take a look at them separately.

- ***What to log?*** - this question is mainly referring to the first point of the best practice rules we listed above. It is a question, what to log on which level, so it helps users maximally understand problems, warnings and runtime behaviour.
- ***Where to log?*** - the other question is about log command placement in a source code. A logging piece of code can be part of each run of a loop on a summary message after it finishes.

- ***How to log?*** - according to [11] is the most challenging question of developing, maintaining high quality and overall consistency of logging code. The publication declares logging anti-patterns as recurrent mistakes which may hinder the understanding and maintainability of the logs.

Up to now we were more focused on how to implement logging in an application code for better understanding of a reader and maximum information content. But we want to design an automated system, so the first step after reading a message in free text form from a log file is to transform it to a format which is understandable for machines and suitable for machine learning.

In literature [19,24] we can find a term **log key**. Log key techniques suggests representing every log entry by a text pattern of constant free-text part of the log message, which corresponds to a print statement in source code, followed by a vector of variables from that message - expecting formatting described in the 4th point of our best practice rules. For further reference we will this vector of variables call **log variables**. This representation also does not work with exact time-stamps and instead of that introduces an extra variable which hold time difference to previous log entry. For better understanding and illustration take a look at Table 2.1.

Original message	Pattern with wildcard (constant part)	Pattern key	Variable vector (variable part)
t_1 Pipeline stated to operate	Pipeline stated to operate	k_0	[$t_1 - t_0$]
t_2 Delivered 5 messages	Delivered * messages	k_1	[$t_2 - t_1$, 5]
t_3 Failed connection to server server-1	Failed connection to server *	k_2	[$t_3 - t_2$, server-1]
t_4 Failing to connect to server-1 for 10 minutes, connecting to server-2	Failing to connect to * for * minutes, connecting to *	k_3	[$t_4 - t_3$, server-1, 10, server-2]
t_5 Failed connection to server server-2	Failed connection to server *	k_2	[$t_5 - t_4$, server-2]

Table 2.1: Example of log transformation to log key format.

Since this structured format is not, exactly, human readable, no programs or applications provide logs in that form, so we should discuss solutions to the problem of transforming a free-text format to more structured format. Traditional solution implemented by operators is based on regular expressions. The problem is, there is no universal database which would contain all regular expressions for every application in every version, so limited databases of that type have to be maintained by individual companies or teams. Fortunately, this afford

can be partially handled by automation instead of pattern development on demand after observation.

The first possibility is source code analysis. Many projects are now released as an open source which make the source code easily accessible. It is possible to write a script which greps all lines related to logging and parses the patterns as was implemented in [63] for Java, C, C++ and Python in 2009. A disadvantage is that the source code has to be known prior log analysis and also not every application provides access to the code.

Another solution for log parsing problem - LCS - is more suitable for streamed data, to which category logging belongs. An algorithm using Longest common subsequence is described in [24] with accuracy higher than 95%. The procedure accuracy is even improved in [18] by prefix trees. The potential problem may be similarities of multiple nodes in same cluster where, for instance, IP addresses may start with same numbers. This is why [31] states that log pre-processing based on domain knowledge can significantly increase log parsing efficiency. It means replacing domains or node names, which may start with environment name, by a hash or removing IP addresses.

The last one, we would like to list here, is an approach introduced in Drain [32]. The system is using log message filtering based on number of tokens and a limited number of first tokens which are supposed to be part of constant parts of logs. The potential weakness of this approach is possibility of placing unstable number of tokens as one variable, but it is not often observed behaviour.

We can say, there are many ways how to implement logging as well as ways how to process such logs when there are stored in a file. Used technology will differ form and architecture to an architecture, but the concept we want to present in this thesis has to be as universal as possible.

Table 2.2 shows some currently available framework for log parsing in online and offline form with their features and average accuracy based of a test in [32] and post pre-processing results in [31]. Keep in mind, that both evaluations of average accuracy were performed on different datasets, so the performance differs for same methodologies. Since log parsing is not in the scope of this work, feel free to find more information in LogPai[2] project, which is the state of the art log parsing collection with benchmarch tool.

2.1.2 Machine monitoring

Application logging should give you all potential answers about runtime behaviour of an application, but knowing statistics about a machine you are running the application on may give you important insights. This have been proved by [21] using VM operation logs with machine statistics. There are sever variables which are useful to know:

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

Framework	Release year	Methodology	Training	Avg.acc. [32]	Avg.acc. [31]
LKE [24]	2009	clustering	offline	0.608	0.6625
IPLoM [43]	2009	heuristic based	offline	0.894	0.8825
LogSig [57]		clustering	-	-	0.9425
SLCT [60]	2003	heuristic	-	-	0.9125
SHISO [45]	2013	trees	online	0.772	-
Spell [18]	2016	LCS	online	0.906	-
Drain [32]	2017	trees	online	0.934	-

Table 2.2: Overview of log parsing frameworks, where Avg.acc. stands for Average accuracy.

- **CPU utilization** – Processing power of CPUs is the heart of every computer machine. It can happen that your application consumes all computing resources and then it slows down or even stops working properly. This is the reason why many engineers sets a warning level on CPU usage to be alerted in advance and have time to react. This information and earning can also be beneficial side variable to information contained in logs.

- **Memory usage** - Reading and writing directly to RAM is much faster than swapping to the main drive, so most of applications are trying not to. Current price development allows us to have memory of a size, developers from 90's could only dream about, but the problem is that our usage grows hand to hand with the size we can effort, so we still have to deal with freeing the memory. It is really language dependent, but for instance lets speak shortly about Java, which is one of the most common languages used for commercial software. A Java application where memory management is fully in hands of a garbage collector can reach a point, where memory is full and processes of the collector blocks the application for couple of minutes. It is true, it does not happen often, since memory if being freed continuously, but there are observed incidents.

- **Drive stats** – Since drive size is in general much higher than memory, we do not have to deal with a problem that drive is too small to serve as a backup place for overflowing memory, even though it is recommended to have at least couple of gigabytes free for correct and smooth functionally of a server.

  Occupancy of a drive plays more important role for application which are more big data focused such as databases or message brokers. In case a drive is full, such application can start rejecting new incoming documents which can start a snow-ball effect. One more interesting case from the real industry. If your application logging is not correctly set (too low significance level and no log rotation), it may happen that log files occupies more space than actual data you want to store. At that moment it is even better to know the distribution of drive usage than single percentage number of used space.

Another potentially useful but not definitely crucial information about drive is input and output statistics. Reads and writes per second or number of queued instruction can imply problems, but they would be there probably from beginning.

- **Bandwidth, SNMP, HTTP(S) and ping** - In the age of cloud application and services, the internet plays even more significant role than before. Unfortunately, the internet is also the most unreliable bottleneck of all elements in this list. There are many ways how to monitor the internet, you can have a health check from a remote machine testing attainability or have local tests for connection speed. Any early warning flag is useful.

- **Internal system events** - Every server has its operation system under which roof applications are running. Since OS - application is parent - child relation, there is a high chance that system action cause reaction in application. Another thing is that we can be able to identify user actions and distinguish standard operations from others. So, for example do not trigger an alert after a planned restart.

There could be potentially found more variables which can be tracked, but our list contains the ones which have the most impact according to us. Also every application itself has its own metrics which can be monitored, like response time of an Nginx server or count of request per second to database, nevertheless we cannot list every metric for every application and software.

Also it is necessary to keep in mind, all the metrics do not have to serve just to problem detection. Another important mission of monitoring is providing estimation for dynamic server provisioning and management [12,35] and many others.

2.1.3 Problem detection practise

Architecture owners and operators are investing a lot of time developing scripts and tools which would free them from tracking all log files and time series to keep everything running. But still you can find many companies around which do not implement event the basic principles.

Understanding of software runtime behaviour though logging is getting harder and harder with every new modern architecture introduced. We can identify some main challenges of logging within complex software architectures:

- **Complexity** is closely related to our definition of complex software architecture Definition 1.1. Microservices and subsolutions are more and more dependent on each other and it is not easy to uncover the dependency map.

- **Volume** and amount of logs are scaling as their service grow. To serve millions of users, you need to have hundreds of machines with even more running programs and each of them are generating their own logs. [44] in 2013 worked with large scale service which produced about 50GB of logs per hours, but in Chapter 4 we will show that the amount can be even higher for current standalone service.
- **Noise** and **small amount of anomalies** is an important aspect of any architecture. Operators have to be informed about every actual problematic case, but, on the other hand, alerting every state which is just slightly suspicious can easily be overwhelming and time consuming and in the end such alerting system is a little to useful.
- **Online processing** is undiscussable mandatory feature for every alerting system. If warning are not timely in order and warning are delivered before or at least at the time when it happened, there is really no use for such system but to a kind of post analysis.

The first attempts were realized by implementing scripts which were filtering new lines appended to log files for general key words like *ERROR*, *Failed* or *connection lost* with a following notification system. This is really a general approach, so the next improvement comes with deep knowledge of the infrastructure you are running. After investigating log files which tracked past error, you understand more the application behaviour and you can adjust script filter for more specific word combinations.

The next step is collecting all relevant logs and information to a centralized place [48] (Figure 2.1) with real-time forwarding so they can be examined together and operators do not have to deal with each machine separately. It also allows anybody to analyze past behaviour and compare it to any other moment. This is a moment when a lot of commercial and open-source products come in to play. At this moment there are some commercial players for log collection and analysis.

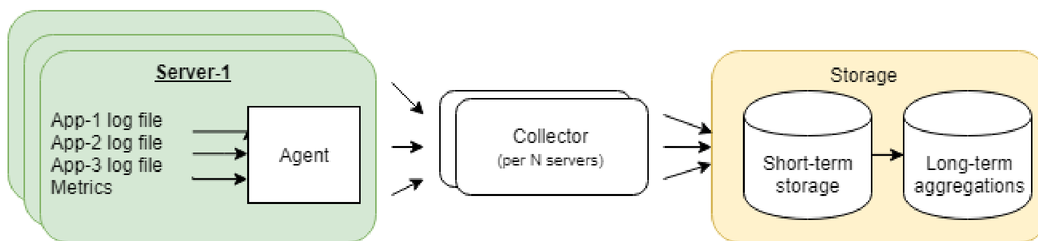


Figure 2.1: Log and metric collection schema.

The main competitors are Splunk[3] Elastic ELK stack[4]. Out of some plugins, integrations and SDKs, both solutions have nearly the main functionality. The main difference is the

[3] https://www.splunk.com/en_us/solutions/solution-areas/log-management.html
[4] https://www.elastic.co/webinars/introduction-elk-stack

format of data which is used for data storing[5]. Splunk storage is based on HDFS [27, 56] style allowing users parse logs and search over them in their raw forms. On the other hand, ELK which is built on no-sql database Lucene, need more work to parse logs before they are stored. Another difference which is more interesting for managers is the pricing, even though Splunk has a small free-tier option, Splunk's general price cannot compete with ELK open-source built.

Another rising star would be Kuberneties monitoring project Prometheus. Docker based architectures are becoming more and more popular so integrated logging solution replacing its old GELF protocol is a step the right way. But we also would like to give a quick shout out to all open-source monitoring software like Nagios, Icinga, OpenNMS, Cacti or Zabbix[6]. All of them are providing excellent support for your application and server monitoring with advance statistic features, but we do not have the resources to cover all of them.

Another approach for problem detection involves E2E topology which stands for end-to-end testing. In short, you are sending all possible test inputs and testing if final output is according your expectancy. This testing is well-working in isolated environment, but in real world case, E2E can suffer to the same problem as your application itself like an internet related errors.

2.2 Anomaly detection algorithms

As has already been said, anomaly detection is heavily studied field cross multiple research disciplines. It means there are many machine learning methods which we can work with, but not every models are exactly fitting for our scope of console log messages and infrastructure monitoring.

In this section we discuss several machine learning algorithms which have previously been used for anomaly detection over console log or software monitoring. However, supervised and unsupervised learning is listed and described below, we pay most attention to the field of semi-supervised learning. We decided so, because for supervised learning a complex and labeled dataset is needed, which creation is really expensive, and there would have to be a designated dataset for every application version and possible architecture setup. Then unsupervised learning did not prove to be efficient, because it has an unspoken requirement of balanced training samples (similar amount of positive and negative samples), which is, generally, not the case for anomaly detection field. Because of that complications, most of recent research works implements a semi-supervised learning which expects only normal behaviour in learning phase. All algorithm are later summarized in overview Table 2.4.

[5] https://devops.com/splunk-elk-stack-side-side-comparison/
[6] https://geekflare.com/best-open-source-monitoring-software/

We also have to discuss possible representation of input for machine learning models. The first one used is called **log key count matrix** which examples can be seen in Table 2.3. The main advantage of that approach is its simplicity in use and computation as well as ability to compress a log period into rather small matrix. It can easily detect logs with potential problematic changed occurrence frequency, but it does not take in account the order of log events as it happened in the time.

In computer logs context, we can expects that there will be a lot of same logs of low significance and the most influencing log keys will not appear so often. Therefor [40] suggests **IDF-based event weighting** along with **contrast-based event weighting** later denoted as IDF-CB weight. IDF-based event weighting - for us more fitting IDF-based log key count weighting - is calcualted as:

$$w_{idf}(k_i) = \log \frac{N}{n_t + 1} \tag{2.1}$$

, where N is the total number of log key windows and n_t stands for number of log key windows in which k_i appears. And contrast-based event weighting is expresed with:

$$w_{contr}(k_i) = \begin{cases} 1, & \text{if } t \text{ appear in training phase} \\ 0, & \text{otherwise} \end{cases} \tag{2.2}$$

Then the final weighting function looks like:

$$w(k_i) = \frac{1}{2} Norm(w_{idf}(k_i)) + \frac{1}{2} w_{contr}(k_i) \tag{2.3}$$

, where *Norm* is the normalization function to scale value between 0 and 1.

The later represented models keep log input in format of a sequence pro preserve the information in which log event followed, but the problem here is in more complex representation for most machine learning models. Some models expect numerical matrices so a sequence has to be represented by a sliding ones matrix which size will grow rapidly based on used window size or session length.

Log key sliding window (sequence)	Log key sliding count matrix
$k_0\ k_0\ k_1\ k_2$	[2 1 1 0 0]
$k_0\ k_1\ k_2\ k_3$	[1 1 1 1 0]
$k_1\ k_2\ k_3\ k_1$	[0 2 1 1 0]
$k_2\ k_3\ k_1\ k_4$	[0 1 1 1 1]
$k_3\ k_1\ k_4\ k_4$	[0 1 0 1 2]

Table 2.3: Log key sliding count matrix example with window size 4 and log key cardinality 5.

So far we spoke only about representation of log keys which stands only for constant parts of log messages and were forgetting about log message variable parts. Most of papers do not work with log variables at all and only some most recent methods are using them as the second level of anomaly detection. The reason is the unlimited space of log variables compare to strictly limited space of constant log parts which come from application source code.

We are also covering time series analysis since many machine variables can be studied in a form of a time series, but since it is not within scope of this thesis, this part is only a brief overview of its potential.

2.2.1 Supervised anomaly classification

Supervised classification models are trained on labelled datasets with both normal and abnormal behaviour. With more labelled data supervised anomaly detection is more accurate, unfortunately, such datasets are rare and if created by operators, they can be hard to maintain with complexity even higher than maintaining database of regular expressions for log patterns as we discussed in Section 2.1.1, but still it is worth to discuss these algorithms and models.

2.2.1.1 Support vector machine

Support vector machine (SVM) is a favourite machine learning concept for classification and regression. This learning model is using hyperplanes to separate two classes defined as a primary minimization problem:

$$\underset{(w,b)}{\operatorname{argmin}} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}\xi_i \tag{2.4a}$$

subject to:

$$y_i(w \cdot x_i + b) \geq 1 - \xi_i, \text{ for } \forall i = 1, \ldots, N \tag{2.4b}$$

$$\xi_i \geq 0, \text{ for } \forall i = 1, \ldots, N \tag{2.4c}$$

And a dual problem representation:

$$\underset{\alpha}{\text{argmax}} \Big\{ \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N} \alpha_i \alpha_j y_i y_j x_i \cdot x_j \Big\} \tag{2.5a}$$

subject to:

$$\sum_{i=1}^{N} \alpha_i y_i = 0 \tag{2.5b}$$

$$0 \leq \alpha_i \leq C, \text{ for } \forall i = 1, \ldots, N \tag{2.5c}$$

As (2.4) is defined, SVM would not be able efficiently learn on linearly inseparable data, therefore kernel trick is used. The idea is based on ability of mapping an original space into higher-dimensional feature space so data becomes separable. It mean we have to have a mapping function for this task $\Phi : x \to \varphi(x)$ and for solving SVM dual problem from Equation 2.5 we need only inner product function of two vectors $K(x_i, x_j) : \varphi(x_i) \cdot \varphi(x_j)$ which is also called kernel function. Then SVM dual problem using kernel function with the same conditions looks like:

$$\underset{\alpha}{\text{argmax}} \Big\{ \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N} \alpha_i \alpha_j y_i y_j K(x_i, x_j) \Big\} \tag{2.6}$$

There are 3 most popular kernel functions [59]:

$$\text{d}^{\text{th}} \text{ Degree pol.} \qquad K(x, x') = (1 + x \cdot x')^d \tag{2.7a}$$

$$\text{Radial basis} \qquad K(x, x') = exp(-\gamma \|x - x'\|^2) \tag{2.7b}$$

$$\text{Neural network} \qquad K(x, x') = tanh(\kappa_1 x \cdot x' + \kappa_2) \tag{2.7c}$$

Robust SVM was used in [36] and later an SVM model using radial basic kernel function and 5-fold cross validation was employed in [39] for failure prediction with in a certain future period (more discussed in 2.2.1.4). The work points out rather long learning time compare to quick evaluation. It means that updating of a model is not possible and it is necessary to retrain it from scratch. Also SVM is not suitable for further anomaly investigations.

2.2.1.2 Decision trees

Tree structure-based models are the most intuitive and easy to use machine learning methods. Decisions are based on attributes with the best information gain, so we can image such model as a tree with conditions in nodes and final classification in leaves. For example, in [13] a decision tree model is employed to diagnose failures in request logs.

In Figure 2.2 we can see a decision for trained data from trained example from that work. The figure illustrates 15 failures of 20 records associated with machine x and 10 error requests of 16 records for Request types y.

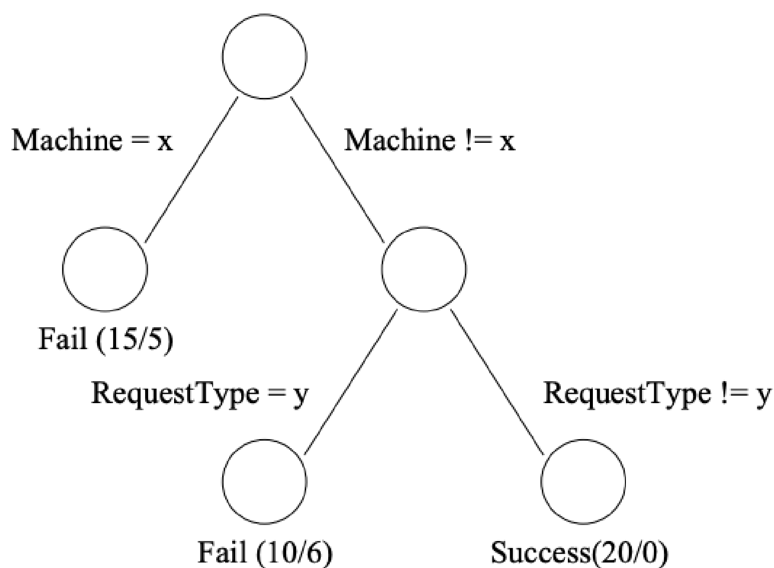


Figure 2.2: Example decision tree anomaly detection from [13].

So, intuitively, adjustations to the model do not have to be done with full retraining as it is in the case of SVM models, but just by adding new conditions replacing the original leaves of the model followed by new classification leaves. Also further investigation is easy, trait forward and easy to explain.

To be more formal, decision tree models are learned to maximize gain function over input labeled features. Such gain function may differ cross fields of use, so for instance one of the original gain functions used in [13, 53]:

$$Gain(x_i, t) = H(t) - H(x_i, t) \tag{2.8}$$

, where $H(t)$ denotes the binary entropy at node t, and $H(x_i, t)$ is the sum of entropy of child nodes after making the split based on feature x_i. And then [13] introduces a new gain

function which respects limited possible values and probability of a problem on particular server:

$$P(x_i = j; t) = \frac{\text{\# of failed requests at node } t \text{ with } x_i = j}{\text{\# of failed requests at node } t} \quad (2.9a)$$

$$Gain(x_i, t) = -H(P(x_i; t)) \quad (2.9b)$$

, where $P(x_i = j; t)$ represents a multinomial distribution over values of x_i.

2.2.1.3 K-Nearest Neighbors

K-nearest neighbors model (kNN) typically bases its decision on strict geometrical distances, it means, it is useful mostly for sliding window count matrix approach since spase matrix of sliding one feature expression would not describe such distances properly. Also [39] proposes using double thresholded limited kNN also called bi-modal nearest neighbor. Such kNN decision process is based only on samples in range $d_1 < x < d_2$, where d_1 and d_2 are estimated based on training sample.

2.2.1.4 Supervised model comparison

We describe SVM, decision tree and k-NN model in this section and all this methodologies were compared in [39] along with rule-based classifier - Ripper. Figure 2.3 displays f-measure, precision and recall for predicting a problematic state within next 1/4/6/12 hours on BlueGene/L dataset (described in Section 4.1). Classification was based on multiple sliding count windows on different metrics and also a log key sliding count matrix of length of 4/16/24/48 hours respectively.

We can see that results for short future periods are not really so good and overall performance is dramatically increasing with with increasing size prediction window. With a prediction window of high size it is much more likely to hit a real recorded fatal error, but also it dramatically decreases value of such prediction

2.2.2 Unsupervised anomaly detection

Compare to supervised methods which need large labelled datasets to train their model, unsupervised methods work with data which do carry any prior information and the models are based only on their statistical attributes. As it has already been said, unsupervised methods are more promising for real-world implementation since there is lack of labelled data, but there are problem with unbalanced datasets, which is, generally, our case.

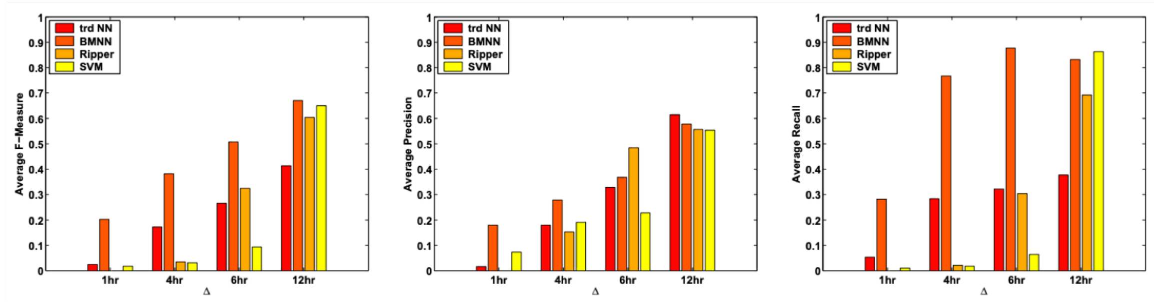


Figure 2.3: Supervised method comparison of f-measure, precision and recall from [39] on sliding window feature matrices of length 12 hours. Figure was composed from Figure 3 in original paper. Trd NN stand for standard Nearest Neighbor, BMNN refers Bi-Modal Nearest Neighbor, Ripper is a rule-based classifier and SVM is a Support Vector Machine model.

2.2.2.1 Log clustering

K-means ([42]) is a popular clustering algorithm which goal is finding k clusters and their centroids $(C_i, c_i) for i = 0, \ldots, k$ minimizing equation (in this case based on l_2 norm):

$$\underset{\forall (C_i, c_i)}{\operatorname{argmin}} \sum_{x \in C_i} \|x - c_i\|_2^2 \tag{2.10}$$

Once we are able to cluster log key count matrix from non-label dataset, it is possible to classify only the main representative of each cluster and then compute to which clusters further observations belong.

[40] proposes two-phase cluster learning – initialization and online learning. This procedure works with weighted event/log key count matrix (as described in the beginning of Section 2.2) and then Agglomerative hierarchical clustering is used for the initialization and creation of two clusters – normal behaviour and anomalies. The second phase od learning serves for cluster further adjustment, new observations are either added to a cluster if distance is smaller than a threshold or a new cluster is created. The work claims around 60% average precision in their experiments.

2.2.2.2 PCA

Principal component analysis is an old and powerful statistical approach proposed in 1901 by Karl Pearson [23]. The main goal is to transform an original feature space into a feature space with smaller dimension maximizing variability of the original data. It helps us identify similarity between data and identify possible outliers by preserving major characteristic.

The same principal can be applied to log analysis. After applying PCA to log key count matrix all standard behaviour will be aligned along the first several principal components

and on the other hand anomalies will take place fare from them. [63] projects a single log key count vector y according following equation:

$$d_y = (1 \breve{} PP^t)y \tag{2.11}$$

, where $P = [v_1, v_2, \ldots, v_k]$ is a matrix of the first k main components. Then if the distance d_y is bigger than a threshold, state represented by vector y is classified as an anomaly.

The paper selects the number of principle components enough high to cover 95% of data variance and the threshold limit based on square prediction error with limit of 0.001. With increasing both variables we can decrease the number of false positive alarms but with cost of misclassification of some anomaly states. The proper setting is highly case sensitive and has to be estimate by experimenting.

Another problem which comes with PCA is, the methodology is sensitive to the data used for training and is not easily adjustable based on additional input. Also as we see in (2.11) log key count vector of a specific size is expected, which makes it harder to record events/log keys which were not present in the training data. We would rather recommend using PCA model for post problem analysis purposes.

2.2.2.3 Invariant mining

Invariant mining was for a long time one of the most promising model for log anomaly detection. The intuition behind software invariant comes from software determinism, expecting, there are just a specific amount of ways to get from a state of a software to another one. Ultimately, invariant mining can recognize a linear dependency between logged actions in normal process execution.

To illustrate this problematic, take a look at simple process flow of a program in Figure 2.4. We can see that every process starts with establishing connection to a server. This part of an execution contains two different log keys, exactly one which says where it is connecting (log A) and unspecified number of messages showing the task has not been completed (log B). Than after successful connection to a server, process retries a variable from it and based on its value process continues differently. The common to every normal execution is log announcing the connection has been successfully closed (log G).

For our case of a simple process flow, if the number of logs A is the same as the amount of logs G, there was not an abnormality in whole execution. But, for instance, in case $sum(A) \neq sum(G)$ and $sum(A) = sum(C)$, we know that connection was successful and there was a problem further in the process execution. An advantage of invariant mining is its independence of specific executions since it searches for common rules. Intuitively, the same case would be, for instance, file open and close, connection and disconnection from

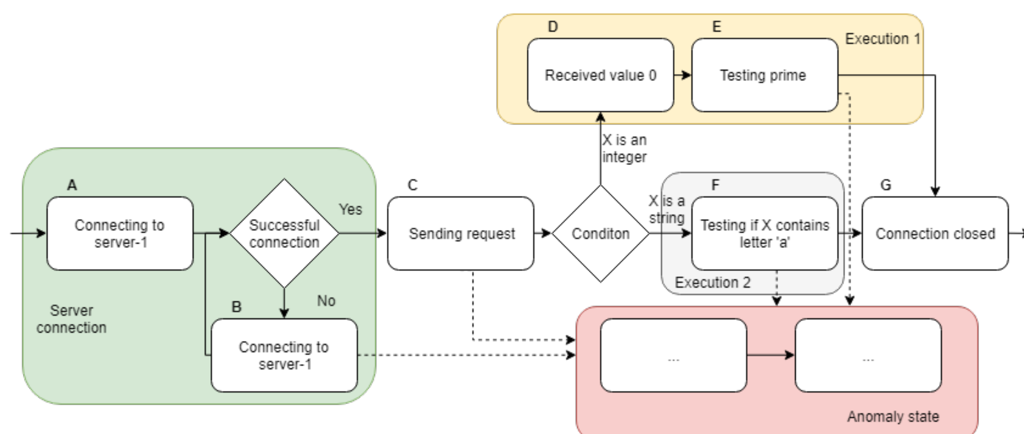


Figure 2.4: Simple process diagram. Dashed connection stands for potential fatal exceptions.

a database or message about timestamp parse failure followed by message about applying default timestamp.

The first application of invariant mining is in [41] which uses singular decomposition on log key count matrix to estimate the number of invariant, then it is followed by brute force mining algorithm to find potential linearity. As final invariant are marked only the couples which appear in more than 98% of relevant event/log key count sessions.

Invariant mining is classified as an unsupervised method, but it could be placed to class of semi-supervised methods because it can, potentially, achieve much butter results if it is trained on data which do not contain any anomaly. In case, morer than 2% of all training data are anomalous sessions, the original algorithm would not yield any invariant. It is possible to decrease the threshold and take number n, estimated by the decomposition step, most frequent invariant, but it is again a trade-off between quality and rapidly increasing potential false positive rate.

Even though invariants can be easily added if an adjustment is needed, invariant mining does not carry any information about event which happens between the related actions. It determines invariant mining to work mostly with session windows. On the other hand, context execution provides a good reasoning behind a classification, which makes it easy to understand.

2.2.3 Semi-supervised anomaly detection

Up to now, we discussed methods which were either trained on labelled or unlabelled data containing normal and abnormal behaviour, but semi-supervised learning is expecting only normal (one class) behaviour in training set. We understand, having strictly normal behaviour can be considered as a labelled dataset which we said to be hard to create and maintain, but to create a labelled dataset with most of potential anomalies present and

labelled in much harder than having a clear execution since a stable service does not experience problems more often. From this point of view, semi-supervised models are the most promising for implementation in real world scenarios.

2.2.3.1 One-class support vector machine

OC-SVM [54] is an anomaly detection adaptation of supervised SVM (Section 2.2.1.1) with the difference, the object of OC-SVM to is finding a maximum margin which separates training data from the origin. The problem is defined by following quadratic equation:

$$\underset{w,\xi,\varrho}{\text{argmin}}\ \frac{1}{2}\|w\|^2 - \varrho + \frac{1}{\upsilon N}\sum_{i=1}^{N}\xi_i \tag{2.12a}$$

$$\text{subject to}$$

$$w^T\phi(x_i) \geq \varrho - \xi_i,\ \text{for}\ ,\forall i = 1,\dots,N \tag{2.12b}$$

$$\xi_i \geq 0, \forall i = 1,\dots,N \tag{2.12c}$$

, where ϕ is a kernel function. This equation can also be rewritten in form:

$$\underset{w,\xi,\varrho}{\text{argmin}}\ \frac{1}{2}\|w\|^2 - \varrho + \frac{1}{\upsilon N}\sum_{i=1}^{N} max(0, \varrho - w^T\phi(x_i)) \tag{2.13}$$

Further more, [46] suggests kernel approximation with random Fourier features to address the scalability problem of kernel machines replacing standard kernel functions presented in Equation 2.12 as ϕ with mapping based on Fourier transform of the kernel function, given by a Gaussian distribution::

$$p(\omega) = \mathcal{N}(0, \sigma^{-2}\mathbb{1}) \tag{2.14}$$

, where $\mathbb{1}$ is an identity matrix, and then such mapping function looks like:

$$z(x) = \sqrt{\frac{1}{D}}[\cos(\omega_1^T x)\dots\cos(\omega_D^T x)\sin(\omega_1^T x)\dots\sin(\omega_D^T x)]^T \tag{2.15}$$

But we have not found any work which would be using OC-SVM for console log anomaly detection, we cannot state any performance comparison here, but we can expect the same positives and negatives as we listed for standard SVM in Section 2.2.1.1.

2.2.3.2 Workflow inference

Workflow inference is natural follow up to invariant mining. CSight [6] is an example of a framework which is half-way through to full workflow inference method. It mines invariant rules and use them as input to counter-example guided abstraction refinement [15], so a basic workflow improves to a state in which it satisfies all of mined invariant. The result model is a communicating finite state machine [9] and every execution which does not fit to the automata behaviour is classified as an anomaly.

The main step back of CSight is its requirement for users to select log pattern to consider, which again requires user domain knowledge and unnecessary interaction. As well as that training data have to contain multiple similar runs so invariants can be mined, but this is a problem inherited from invariant mining methodology.

The state of the art of workflow inference problematic is presented in CloudeSeek [37]. It analyses log keys ordering in a log file and compresses it to multiple automata for different processes. Automaton walk-through is triggered with every newly incoming log key and classified normal if at least one automata visited all states, with the difference to a classic walk-through, the process can continue from any already visited state. In case there is a transition which is not satisfied by any edge of automata, anomaly alert is triggered.

We are using an example case from the original paper to illustrate training process and classification of CloudSeek. Let us consider two log key sequences in training set: k_1, k_2 and k_1, k_3, which can be seen as that k_1 precedes k_2 and that k_1 precedes k_3. This behaviour be represented as an automaton displayed in Figure 2.5, with starting states for k_x being a collection of target states of edges labelled as k_x.

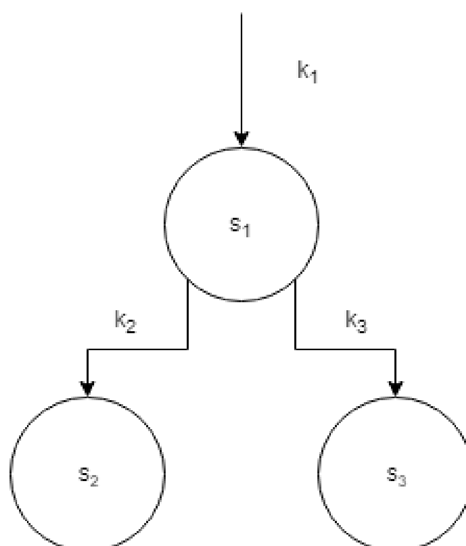


Figure 2.5: Sample automaton to illustrate workflow inference anomaly detection process.

For the classification, we first consider a sequence k_1, k_2, k_3. The walk-through triggered on k_1 starts in state s_1, then it is followed by k_2 which sets the set of states to continue to s_1, s_2. The next incoming is k_3, with this log key the walk-through can proceed from s_1 to s_3, so no anomaly is identified and since all state have been visited walk-through ends. The other case happens for a sequence k_2, k_1, k_3. Starting in state s_1 there is no edge we can move having following log key k_1 and anomaly warning is yield.

Workflow inference is, as well as invariant mining, on the edge between unsupervised and semi-supervised methods. The best result is achieved with training data of just normal behaviour, because then there is no chance, an automaton contains an edge which would satisfy anomaly behaviour. In case of training data with both behaviour types, a threshold for creating edges has to be set to avoid normalizing abnormal executions. Another possible for anomaly detection is heuristic based automata state classification, for instance error log keys. Then a warning can be also triggered earlier if the only possible walk-through leads to such state. On the other hand, automates are rather easy to modify based on additional inputs, which can make it easy for operators to adjust the model according their needs.

2.2.3.3 Long short-term memory

LSTM [34] model, an instance of recurrent neural networks (RNN), implemented in DeepLog [19] is the first neural network model we are discussing here and is currently the best solution to anomaly detection for software logs. The intuition behind that approach is based on natural language processing (NLP), but instead of studying the content of separate log messages, it focuses at log key sequences with log keys instead of words. In every language we can identify rules and grammar and LSTM-based model expects the same for log sequences. In the end, we can see a log file as a dialog between software and operators.

The previously named work also implemented an anomaly detection based on N-grams, but LSTM proved to be more efficient and promising, so we discuss only the neural network approach here. General LSTM takes a vector of values and return probability distributing for values which can follow the input vector. An a sequence is classified as anomaly, if probability of last log keys of then sequenceis not between the first n log keys with highest probabilities. Unfortunately, neural networks are more complex for formalization, so we are including formalization specific for log analysis and a Figure 2.6 of used model from [19].

Lets expect we have a set of all n possible log keys $K = k_1, k_2, \ldots, k_n$. Then for training and classification we will be always working sliding windows sequences w at a time t of length h denote as $w_t = [m_{t-h}, m_{t-h+1}, \ldots, m_{t-2}, m_{t-1}]$, where m_i stands for i^{th} log key of an input sequence of log keys. Having all this set, we can formalize model output of a conditional probability distribution as $Pr[m_t = (k \in K)|w_t]$.

In the Figure 2.6 we can see that single m_i of w_t is an input to a single LSTM block and each block of a level x is also fed by a hidden vector H^x_{t-i} and a cell state C^x_{t-i} from

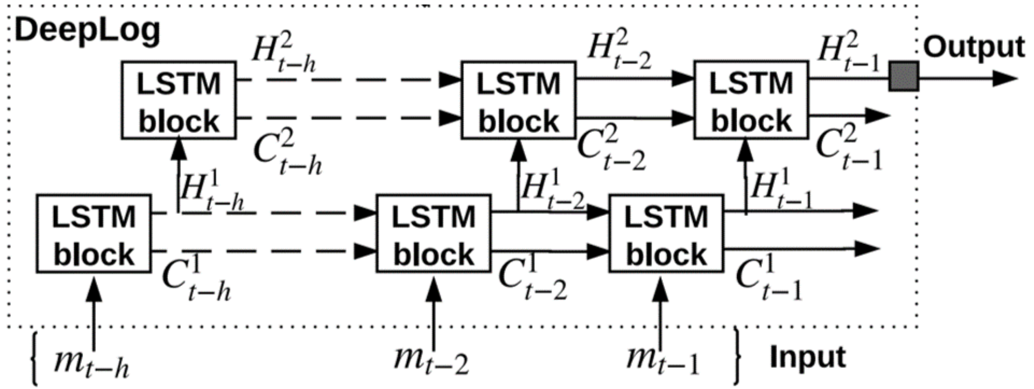


Figure 2.6: Visualization of LSTM used in DeepLog. Image used from [19].

the previous time step (block timely one step older). In this case, this composition is called deep LSTM and deeper layers of the network are fed by sequence of hidden vectors instead of log keys correspondingly.

There are 4 customisable parameters for such neural network decision process: 1) window size h used as input vector, 2) number of layers/depth of LSTM L, 3) amount of memory for each LSTM block α and 4) threshold g saying how many highest log key probabilities are considered to be normal. The default setting of DeepLog is $h = 10$, $L = 2$, $\alpha = 64$ and $g = 9$.

The framework is focusing on log key sequences and log key parameters separately using the same infrastructure. There is also a built-in functionality for feedback, which is adjusting weight within the neural network if an operator classifies alerted behaviour to be normal. This helps to improve the model with normal situations which were not present in the training set.

DeepLog can capture nonlinear and high dimension dependencies of system normal execution with high accuracy, but since neural networks works as a black box for users, there is just a little chance to provide more than alerting to operators. This is the reason why DeepLog also builds workflow model together with LSTM to give operator change to identify the root cause of the problem alerted by neural network.

2.2.4 Comparison of anomaly detection models

All below listed algorithms are compared in Table 2.5. In the table you can also see efficiency of algorithms defined by precision, recall and f-measure. The values are taken from both papers for HDFS data and one for BGL data. We can see that for HDFS data supervised methods SVM and decisions trees perform really well, but for BGL, which has more complex execution and all anomalies are not present in training set, the performance lacks in recall and F-measure. From the unsupervised section we can see that invariant mining outperforms

Algorithm	Type	Model update	Classifying
SVM	Sup.	Retraining	Online
Decision tree	Sup.	Adjustable	Online
Clustering	Unsup.	Retraining	Online
PCA	Unsup.	Retraining	Online
One class SVM	Semi-sup.	Retraining	Online
Invariant mining	Semi-sup.	Adjustable	Sessions
Workflow inference	Semi-sup.	Adjustable	Online
LSTM	Semi-sup.	Adjustable	Online
Autoencoder	Semi-sup.	Adjustable	Online

Table 2.4: Anomaly detection algorithm overview.

log clustering and PCA for both datasets in [33] but the later work proved superiority of neural networks. Both experiments use rather small training set in comparison to test set we speak about 1-2%.

Algorithm	Efficiency [33] HDFS	Efficiency [33] BGL	Efficiency [19] HDFS
SVM	1-1-1	0.95-0.57-0.71	-
Decision tree	1-1-1	0.95-0.57-0.72	-
Clustering	0.87-0.74-0.8	0.42-0.87-0.57	-
PCA	0.98-0.67-0.79	0.5-0.61-0.55	0.98-0.67-0.79
Invariant mining	0.88-0.95-0.91	0.83-0.99-0.91	0.88-0.95-0.91
LSTM	-	-	0.95-0.96-.0.96

Table 2.5: Machine learning algorithms used for anomaly detection with their efficiency describe by precision, recall and f-measure using [33] and [19].

2.2.5 Time series analysis

As we discussed in Section 2.1.2, a general server monitoring system produces mostly numeric time series, which are fitting to the scope of time series analysis. Being efficiently able to predict high workloads, performance peeks and gaps can limit expenses and also limit possible delays. There is a high requirement from companies to be able to scale up and down their services according current demand to take it maximally cost-effective. According to [5], 40%-50% of cost of running a high-performance computing platform is in its energy consumption.

We would like to list some time series anomaly detection algorithms which are frequently used:

- **ARMA models** - Autoregressive moving average models were introduces in 1954 by [30] and since the time there are many adaptations of the original work. ARMA model is represented and defined by order of its autoregression and moving average parts, but the main problem is it assumes stationarity of input series, but this is too strong assumption for real world data and also for variables which we can obtain from machine monitoring. **ARIMA** (AutoRegressive Integrated Moving Average) is the first generalization of the standard ARMA model to cover non-stationary numeric series introduced by [26] in 1970. Later **SARIMA** - (Seasonal ARIMA) - was introduced to deal with seasonality and **ARFIMA** [4, 38] - (Autoregressive Fractionally Integrated Moving Average) - for long-memory processes allowing fractional degrees of integration.

- **Singular Spectrum Analysis** or Singular Value Decomposition (SVD) are often used to de-noise univariate time series or to study their spectral profile. The aim of SSA is to decompose a time series into a sum of a small number of interpretable components such as a slowly varying trend, oscillatory components and superimposed noise interferences. - citation from [58] and example of usage on monitoring data can be found in [38].

- **Hierarchical temporal memory** (HTM) is a promising neural network model for time series based on tree-like hierarchical structure. [3] employs HTM for anomaly detection and reachich score around 65 in NAB dataset (discussed in Section 4.1)

- **Autoencoder** neural network belong to an unsupervised class model group with a goal of dimension reduction and minimal correlation lost between the original features. [8] uses autoencoders to filter noise and later determine anomaly presence based on reconstruction error.

2.3 Anomaly detection evaluation

Model evaluation methods are important not only for final model selection and comparison, but servers also for model parameter tuning and performance metric estimations. The basic evaluation is based on a confusion matrix as the one you can see it Table 2.6 which was created to fit our case of anomaly detection. There are 4 different combinations of predicted label vs actual label: **TP** - True positive, **FP** - False positive, **FN** - False negative and **TN** - True negative, which counts are used for other metrics:

- **Accuracy (ACC)** - a ratio of correctly predicted observation and the total observations. This measure cannot be used for our unbalanced case, since optimizing this measure would cause clasifing all behaviour as normal.

		Predicted label	
		A	N
Actual label	**A**	True Positive (TP)	False Negative (FN)
	N	False Positive (FP)	True Negative (TN)

Table 2.6: Confusion matrix for anomaly detection, where A stands for anomalous behaviour and N for None and consequently normal behaviour.

$$ACC = \frac{TP + TN}{TP + FP + FN + TN} \tag{2.16}$$

- **Precision (PR)** - a ratio of correctly predicted positive samples, in your case anomalies, to the total number of positive predictions made. It addresses the question of have many triggered anomaly notifications were confirmed as anomalies. For unbalance cases as we are focused at (2.17) could have also weighted form to add significance to TP predictions.

$$PR = \frac{TP}{TP + FP} \tag{2.17}$$

- **Recall (RC)** - also known as sensitivity, is a ratio of correctly predicted anomalies to the all actual anomalous behaviour. This measure expresses how many of actual problems a model was able to detect. For our case, if we want to detect all problem, RC has to be equal to one.

$$RC = \frac{TP}{TP + FN} \tag{2.18}$$

- **F1 score** [14] - a harmonic average of Precision and Recall. Therefore, this score works with both false positives and false negatives and better represents unbalanced data samples. From the so far named measures, F1 score is the most suitable for anomaly detection evaluation.

$$F1 = \frac{2 * (RC * PR)}{(RC + PR)} \tag{2.19}$$

Precision, recall and F1 score were used as main measure for instance in [19, 40]. But [36,46] suggest using a ROC curve eventually AUCROC or an alternative in form of AUCPR curve [17].

- **AUCROC** - Receiver Operating Characteristics curve (ROC) is a ratio between true positive rate and false positive rate and it expresses much is a model capable of distinguishing between classes. Therefore maximizing area under the ROC curve (AUCROC) is find a model with best distinguish potential.
- **AUCPR** - Precision-Recall curve is a ration between precision and recall and it express the trade-off between the true positive rate and the positive predictive. [51] states, that AUCPR is more useful for imbalanced cases, as the one we are working with, since it takes in account both TPs and FPs and TN cases are not interesting for anomaly detection problem

Unfortunately, utilization of these curves is really limited since we are working with training set containing only normal behaviour. It means we would have to have all training set classified as normal behaviour, maximum decision bound has to be selected anyway and there is not much space for parameter tunning.

There is also possibility of anomaly detection evaluation in non-labeld dataset, which we would like to also, briefly, mention, but since this is not our case, it servers more for complexity of this section. [28] suggest using Excess-Mass (EM) and Mass-Volume (MV) curves with feature sub-sampling and aggregating to overcome original high-dimension limitations.

Chapter 3

Problem definition

This work aims to address a need for automated anomaly detection and prediction within complex software architectures. The goal is to be able to detect every anomalous behaviour and keep level of false alarms low. Using terminology from Table 2.6 FN count should be low and preferably equal to zero and also secondary minimize FP count. This problem is formalized as Neyman-Pearson task or Neyman-Pearson classification [55] as search for optimal strategy q^* with given conditional probabilities $p(x|D)$, $p(x|N)$ and unknown prior probabilities $p(D)$ and $p(N)$:

$$q^* = \underset{q:X \to K}{\operatorname{argmin}} \sum_{x:q(x) \neq N} p(x|N) \tag{3.1a}$$

$$\text{subject to:}$$

$$\sum_{x:q(x) \neq D} p(x|D) \leq \bar{\epsilon}_D \tag{3.1b}$$

, where D is a set of dangerous/anomalous states, N is a set of normal states, then the whole state space is $K = D, N$, X is a set of observations and $\bar{\epsilon}_D$ is a top bound on danger state classification error from range $< 0, 1 >$. We did not find this formalization used in any other research work related to anomaly detection in a software environment, so we include it here, formalize problem we are solving.

We have to also expect additional noise in software execution because every architecture is under continuous maintenance. Generally, we can expect three types of maintenance jobs: 1. job which does not leave any significant footprint as posting changes to cluster setting, 2. jobs which cause a small disruption to normal behaviour but in short time architecture is stable again and 3. jobs which went wrong and caused a major problem. In case, anomalous behaviour is detected during a maintenance job, operators need to be informed only in the third case of maintenance job, which means a model has to learn the small disruption as

part of normal behaviour. As an anomaly prediction during maintenance job of type 1 of 2 counts as FP as visualized in confusion table for this problem - Table 3.1.

Actual label	Predicted label: A	Predicted label: N
A	True Positive (TP)	False Negative (FN)
M	False Positive (FP)	True Negative (TN)
N	False Positive (FP)	True Negative (TN)

Table 3.1: Problem confusion matrix for anomaly detection, where label A stands for anomalous behaviour, M for maintenance job execution and N for normal behaviour.

Another aspect which we would like to use for model evaluation is throttling for alert notification. It means, there is a limitation of how many anomaly warning may be triggered within a certain time range. For this study, a limitation of one alert warning per 5 minutes was selected, based on the recommendation of experts we worked with. As a result, we can split test execution into 5-minute time ranged and classification of each range is given in the end.

Chapter 4

Dataset

Datasets and correct data interpretation are backbone of every data science project, you can either use it to evaluate your models or even train your models using them. Then the gold mine is labelled dataset which provides the ground truth. The problem is that each dataset is field specific so in our case we cannot use, for instance, a dataset of handwritten digits.

One of the features of data we need is its size. For instance [63] worked with Hadoop cluster logs generated in range of 48 hours by more than 200 nodes with total volume bigger than 200 TB. Unfortunately, this dataset was not shared any further because of a risk or leak of sensitive data since console log data can contain sensitive information which is expensive to filter.

In this chapter, we would like to briefly describe existing datasets, which could be partially used for our case as well as introduce dataset we created to specifically address the use case we were asked to address. The new dataset was called **ELKR** based on software used in tracked architecture and published with this thesis[1].

4.1 Existing datasets

Some existing datasets are rather close to the case of this study, but since we wanted to address a real world problem, we decided to not use any of them and work mostly with our data. We list them here, if anybody else would be interested in using them.

- **BlueGene/L**[2] is a collection of cluster workload logs, error logs and monitoring from years between 1996 and 2005 of a BlueGene/L supercomputer [1]. The overall system consists of 4750 nodes and 24101 processors and all the data are accompanied by information about each problem, where it was detected and resolved as well as

[1] https://github.com/OBorovec/AnomalyDetectionAndPrediction

[2] https://www.usenix.org/cfdr-data#bgp

what was affected. It also contains a simple information about root cause of recorded problems in form of a label from set: Human error, Environment failures, Network failure, Software failure and Hardware failure.

Overall it is very rich dataset, but, unfortunately, at the time we were preparing our experiments the dataset was unavailable, so we could not benefit from it. Still we can see, this dataset was used in papers and experiments of [25, 33, 39, 47] and others.

- **NAB**[3] - Numenta Anomaly Benchmark introduced in [2] contains real-world and artificial time series which can serve for training time series monitoring based anomaly detection of features such we listed in Section 2.1.2 - especially data for AWS cloud watch which covers CPU utilization, network traffic and Drive I/O. For instance time series from section 'Known causes' in the dataset shows that mis-configuration in setting can cause CPU problems and system failure resulting in EC2 request latency.

- **ODDS**[4] - Outlier Detection DataSets collects multiple different datasets for outlier/anomaly detection since 2016 from all the interest fields around. Even though we did not find any dataset which would 100% fit our research, it can be useful for general testing of state clarification of anomalies in general anomaly detection problem setting.

- **Loghub**[5] provides a large collection of system logs per stand alone applications. This collection is cited and used by most of the recent papers and works, which are dealing with system log parsing and analysis.

- **HDFS**[6] contains log from more than 200 Hadoop nodes labelled by a Hadoop expert. There are about 2.9% anomaly behaviour segments along more than 11 millions logged events total. This dataset was also used by many works like [50, 62–64]

4.2 ELKR dataset

As it has been already stated, we would like to address a real world problem, expert architecture operators asked us to solve. Unfortunately, there were not data to work with neither an anomaly detection concept out of a E2E monitoring system. Therefore we needed to design and implement an architecture to collect all generated logs by the core application and additional monitoring of target architecture. The collected information (20-30 GB of data daily) had to be processed, cleared and mask later, so we could publish this dataset.

[3] https://github.com/numenta/NAB
[4] http://odds.cs.stonybrook.edu
[5] https://github.com/logpai/loghub
[6] https://figshare.com/articles/HDFS_Logs/4040124

In this section we describe the architecture from which all data comes, we show main features of this dataset and discuss problem which were recorded.

4.2.1 Service architecture

The target software architecture is built on top of ELK stack[7] with some additional program. There is a list of all applications and software which play the main role in observed architecture functionality and are monitored within dataset:

- **Curator**[8] - an Elasticsearch manager with scheduling and automation support
- **Elasticsearch**[9] - a NoSQL database and search engine -
- **Filebeat**[10] - a file tracking and record shipping tool
- **Kibana**[11] - a visualization software for Elasticsearch with management capabilities
- **Logstash**[12] - a message processing and shipping tool
- **NGinx**[13] - a web server
- **RabbitMQ**[14] - a message-broker software

For further understanding take a look at architecture visualization in Figure 4.1. We can see that each architecture can consist of multiple Elasticsearch clusters and there may be multiple applications running on each server. There is also visible complexity of interconnection between application represented by arrows and connections n-n within each cluster. This dataset contains information from 2 architectures - **Dataset 1** tracks more complex architecture and **Dataset 2** is unique in its size.

Dataset 1 represents architecture not only with 2 cluster of Elasticsearch and RabbitMQ cluster from Figure 4.1, but there is also an additional cluster for another sensitive data, which are stored in files and read locally by logstash instances before the data are sent to Elasticsearch. On the other hand **Dataset 2** architecture corresponds exactly to the visualization, but because it is the second largest production cluster, input traffic and amount of visualizations per second causes different problems. Some general statistic about both architectures represented by dataset can be found in Table 4.1.

To be able to record, store and later examine architecture runtime behaviour, we implemented an extra infrastructure like the one described in Section 2.1.3 based on the same application we track and this infrastructure monitoring is also included within dataset, this

[7] https://www.elastic.co/elk-stack
[8] https://github.com/elastic/curator
[9] https://github.com/elastic/elasticsearch
[10] https://github.com/elastic/beats/tree/master/filebeat
[11] https://github.com/elastic/kibana
[12] https://github.com/elastic/logstash
[13] https://github.com/nginx/nginx
[14] https://github.com/rabbitmq/rabbitmq-server

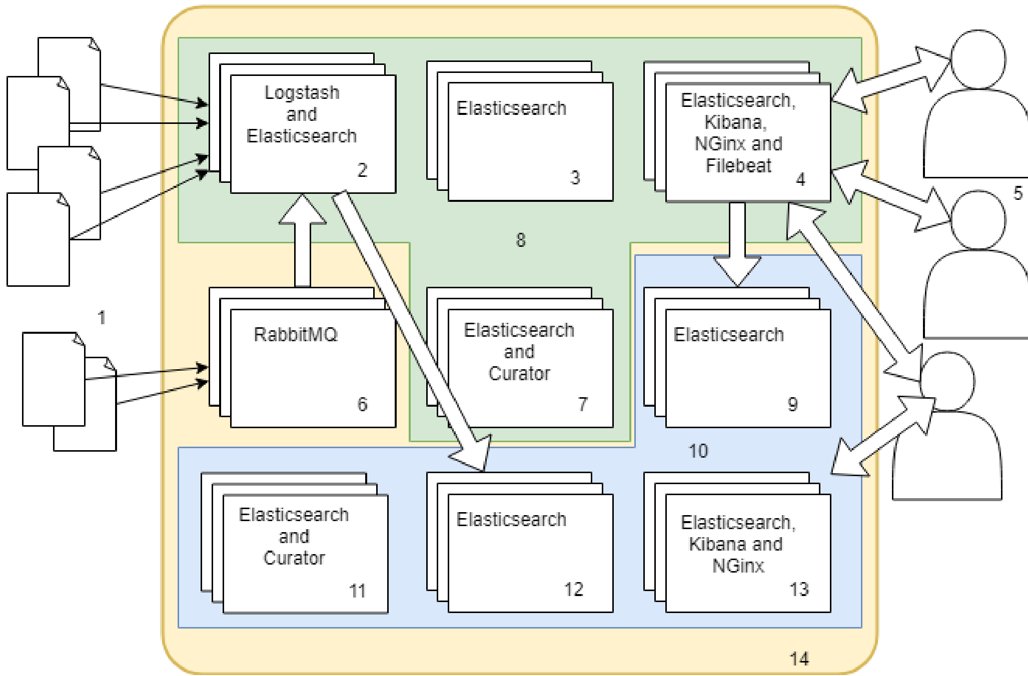


Figure 4.1: Illustration of a logging platform in one environment. Explanatory notes: 1. incoming documents from users, 2. collector nodes with Logstash and Elasticsearch instance, 3. Elasticsearch data nodes of the green cluster, 4. servers running Kibana for log visualisation, 5. users, 6. RabbitMQ cluster, 7. Elasticsearch master nodes of the green cluster of the green cluster, 8. green cluster, 9. Elasticsearch ingests nodes, 10. blue cluster, 11. Elasticsearch master nodes of the blue cluster, 12. Elasticsearch data nodes of blue cluster, 13. servers running Kibana for log visualisation of the blue cluster, 14. logging platform environment setup

	No. of serves	Weekly traffic (GB)	Weekly docs stored (B)	Total stored (TB)	Weekly searches
Arch. 1	76	723	5.05	4	28452
Arch. 2	63	5962	7	30	394789

Table 4.1: Attributes of architectures documented in datasets

is the reason why you can see 5 respective 4 cluster per environment in Table 4.2. We also installed additional native monitoring tools for Elasticsearch, Logstash and Kibana to possibly provide more monitoring features of these software instances.

4.2.2 Dataset structure

We were able to record behaviour of both architectures from generated log messages per application in form of out predefined structure (names of attributes in italics have been masked as a hash, to prevent potential sensitive data leak):

- Common part
  - @timestamp - time of record generation
  - *cluster* - name of cluster from which record came
  - *hostname* - name of server from which record came
  - application - name of application which generated record
- Log record
  - level - severity level of log record (may differ per application - ERROR for Elasticsearch and ERR for Filebeat)
  - logger - name of original logger which produced log record
  - message_pattern - constant part of log message
  - *message_variables* - list of variable parts of log message
- Metric record
  - metrics - list of key-value numeric metrics

Most of the fields above could be generated directly from structure of our log collection system we set, but to create message_patterns and message_variables we had to involve further processing. To benefit open-source nature of software we are interested in, we implemented source code analysis tool according to idea from [63], but source code analysis efficiency was lacking behind identifying only about 60% of all patterns. So as we continue and finished log pattern mining with a clustering approach from [57] resulting to satisfiable level of log message parsing.

As a result we identified nearly 600 different log message types based on their constant parts within our dataset. All of them per application are published along this dataset. The highest cardinality comes from Elasticsearch source code, because this application is the backbone of all architectures we studied as well as its source code is the largest along all listed application.

We also split original logs into two categories - log record and metric record, so it is easier to distinguish between log, which refer to a software state, and a mostly numerical metric. This split is applied for Elasticsearch log messages about slow searches and indexing operations and Kibana and Nging response time messages. This way improves usage of sequence interpretation of dataset since it got significantly cleared from noise. On the other hand time based sliding window still may contain this kind metrics to represent architecture responsiveness, but we recommend to keep your approach consistent and use only the log record parts.

Another part of the dataset is native monitoring of Elasticsearch cluster, Kibana and Logstash in form of nested JSON objects, which contain a lot of interesting measurable attributes like CPU usage, average indexing time, average response time, garbage collection

status, cumulative number of requests and many others (full list in dataset documentation). But these features do not belong to the scope of this work and added for a potential time series anomaly detection research. We involved this collection into our dataset, because we wanted to make it as complex as we are able for further potential research attempts.

Each dataset is split into training and testing phase. In training part, which is first several days of dataset content (Table 4.2), there are no known real problem. Since dataset is created based on a real environment, there are still some maintenance jobs within this period, which may cause small anomalous behaviour typical for successful maintenance, but do not cause a major problem about which we would like to be alerted, as we stated in Problem rules in Chapter 3.

Testing part follows prior training and is annotated with labeled anomalous behaviour. There are also an independent label set for architecture maintenance. Maintenance job is a short execution run by operators - in dataset represented as an one-moment event and can last up to 20 minutes. Job may cause a problem, but ,generally, it causes only a small disruption in normal behaviour which we can see as noise. Therefore alerting FP during a maintenance run should not be penalized since an operator who runs the job can easily mark it as FP as it is in Problem rules mentioned above.

In Table 4.2 you can see information about this labels along with other statistical information about both architectures of dataset. We also add **Dataset 1** training phase visualization in Figure 4.3, full **Dataset 1** visualization in Figure 4.2 and **Dataset 2** visualization in Figure 4.4. Both figures show total and relative amount of records produced per application per 4 hours along with heatmap indication about anomalous behaviour and maintenance execution labels.

-	Dataset 1	Dataset 2
Architecture	Architecture 1	Architecture 2
No. of clusters (ESs + RabbitMQ)	5	4
No. of servers	76	63
Total recorded period	21 days	6 days
Trainig part	4 days	3 days
Log key cardinality	212	558
Log entries	652223	5675122
Metric entries	3682242	4054659
Monitoring entries ES/LGS/KB	725181/ 1629244/ 544034	154161/ 206948/ 207337
Anomalous events	8	4
Anomalous time (of test phase)	3.075%	9.954%
No. of maintenance jobs	141	46

Table 4.2: Dataset attributes

Figure 4.2: Visualization of dataset 1 training phase - a chart of total counts of log and metric records per application per 4 hours in the first part, the second part expresses the same relatively. Under both subplot, there is a visualization of anomaly behaviour and maintenance job labels (red labels an anomaly/maintenance label within the 4 hour range).

4.2.3 Recorded problems

- **A Logstash host not responding** - anomaly detected with New Relict[15] software when a Logstash server is not responding to ping on input port.
- **A Elasticsearch cluster is yellow/red** - anomaly based on internal Elasticsearch cluster health status changed to red or changed to yellow for longer than 5 minutes. Note that this change from green to yellow and back in under of 5 minutes does not mean an anomalous behaviour, since it can be experience anytime a new index is created.
- **Kibana are red** - internal Kibana health status check.
- **NGinx service health problem** - endpoint not fulfilling service metrics in last 5 minutes.
- **Logstash failure** - Logstash not processing new messages detected by ping failure.
- **Snapshot to S3 error** - Elasticsearch index backup failure usualy caused by network traffic issues.
- **Messages cumulate in RabbitMQ** - amount of queued messaged over a safe threshold, which is usually cause by consuming Logstashes problems.
- **RabbitMQ consumers drop** - amount of consumers connected to RabbitMQ cluster lower than predefined number.

[15] https://newrelic.com/

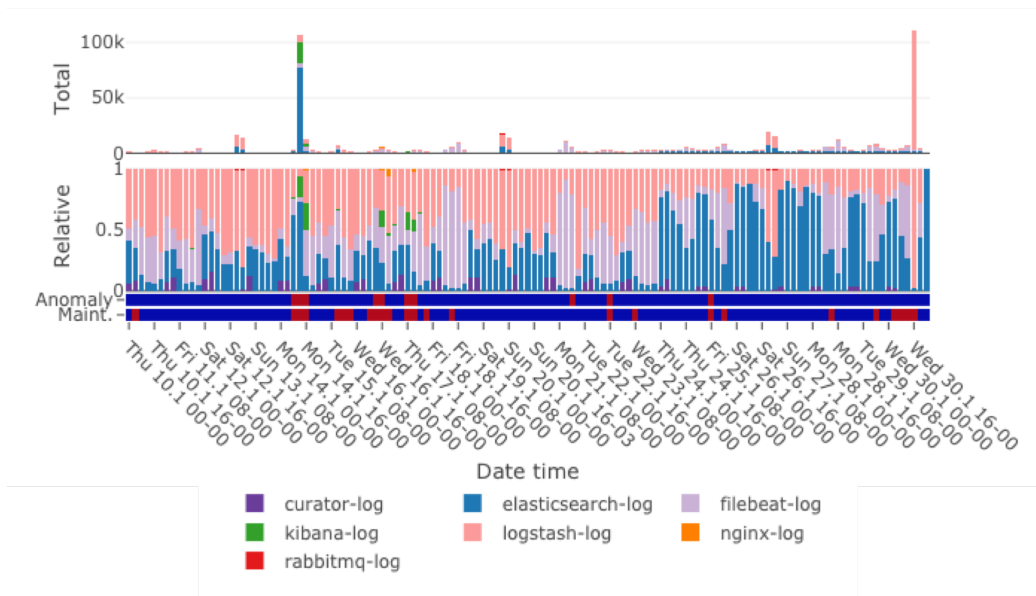


Figure 4.3: Visualization of dataset 1 - a chart of total counts of log records per application per 4 hours in the first part, the second part expresses the same relatively. Under both subplot, there is a visualization of anomaly behaviour and maintenance job labels (red labels an anomaly/maintenance label within the 4 hour range).

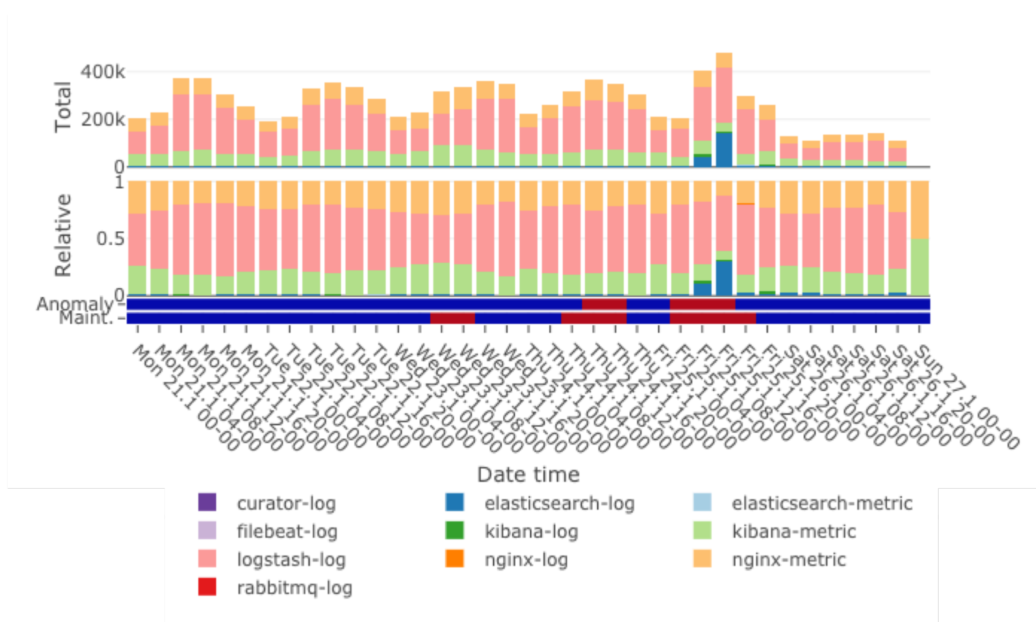


Figure 4.4: Visualization of dataset 2 - a chart of total counts of log and metric records per application per 4 hours in the first part, the second part expresses the same relatively. Under both subplot, there is a visualization of anomaly behaviour and maintenance job labels (red labels an anomaly/maintenance label within the 4 hour range).

Chapter 5

Proposed solution

Workflow-based models like Invariant mining (Section 2.2.2.3) and Workflow inference (Section 2.2.3.2) proved to be efficient in anomaly detection field of software system behaviour with a big added value in form of easy behind anomaly reasoning. Therefore we decided to work with this model type rather than with a fancy neural network model.

The main downside of current workflow-based models, we want to discuss, is their limitation on log key sequences, it means, ignoring all variable parts of each logged record. This limitation exists, because potential space size would increase radically and requirements for training normal execution length could also exceed reasonable limits. We identified two main problems, which we would like to demonstrate on our sample logged execution from Table 5.1, where we expect records $< 1, 10 >$ to be a training part - guaranteed normal behaviour - of the execution and the rest is supposed to be analyzed for anomalous behaviour.

Since Workflow inference model is superior to Invariant mining, we will the problems demonstrate with a Workflow inference model which was trained on all sub-sequences of length 4 of the training part. Such model would represent 7 training sequences by 6 automata of the max depth of 4 which are presented in Figure 5.1.

The first problem comes from studying the problems we listed for our specific case in Dataset Chapter 4. For the "Elasticsearch cluster is yellow" case, we can see, that time distance between specific logs matters for anomaly detection. Timestamp information, eventually time delta, belongs between log variables as was suggested in Section 2.1.1, but since that extra information comes with every logged record type (for us log message, metric and monitoring record), we can work with it on different level than the rest of log variables so we can have a logged record consisting of log timestamp, log keys and log variables

In your given example, automata (c, d, e) of Figure 5.1 contain 2 following log keys k_5 since every automaton execution of Workflow inference model can continue from any previously visited state, these automata can accept any number of following k_5 log keys. Such behaviour is in contradiction with given domain knowledge, if document assigning

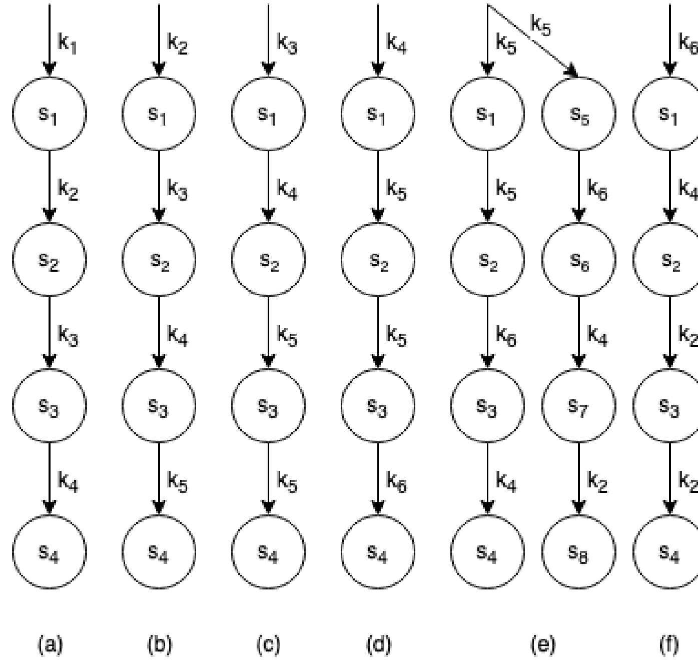


Figure 5.1: Workflow inference model automata for proposed execution from Table 5.1 with training on redocrds $< 1, 10 >$ using sub-sequences of length 4. Letters under each automaton servers for further referencing.

takes longer then 6-time units (time difference between log keys k_3 and k_6), anomaly alert should be triggered. To address this problem would like to introduce a new adaptation of original Workflow inference model called to **Timed workflow inference**, which limits original Workflow automata edge connectivity with thresholded maximum time difference.

The second potential problem comes workflow automata limitation to the only seen sequence. Automaton (b) Figure 5.1 does not contain any other k_2 edge and it would cause anomaly trigger with the first appearance of two following log keys k_2 in testing part. This problem is mostly caused by the small used training set, since this automaton would accept such behaviour if the training set is composed of records $< 1, 14 >$, but it could be also seen as a limitation of this model. Therefore a more relax generalization of Workflow inference algorithm is needed and we propose a graph-based anomaly detection model which we called **Timed graph workflow**. The model generates anomaly probability instead of a boolean label based on irregularity of edge connections. We also suggest a further concept of **Timed hierarchical graph workflow** which effective deals with log variables, but for this thesis, we wanted to correlate with start of the art and did not implement this method.

Record(r_x)	Timestamp(t_x)	Log key(lk_x)	Log key pattern	Log variables(lv_x)
1	0	k_1	server started	[]
2	2	k_2	storing doc * to *	[doc1, index1]
3	3	k_3	unable to assign doc *	[doc1]
4	4	k_4	health from * to *	[green, yellow]
5	5	k_5	assigning doc *	[doc1]
6	7	k_5	assigning doc *	[doc1]
7	8	k_6	doc * assigned	[doc1]
8	9	k_4	health from * to *	[yellow, green]
9	12	k_2	storing doc * to *	[doc2, index1]
10	13	k_2	storing doc * to *	[doc3, index1]
11	14	k_2	storing doc * to *	[doc4, index1]
12	15	k_3	unable to assign doc *	[doc4]
13	16	k_4	health from * to *	[green, yellow]
14	17	k_5	assigning doc *	[doc4]
15	19	k_5	assigning doc *	[doc4]
16	21	k_5	assigning doc *	[doc4]
...	...	...	...	

Table 5.1: Logged record sequence breaking standard Workflow inference algorithm

5.1 Timed workflow inference

The original workflow automata in Workflow inference model were represented by a set of directed graphs with edges in form of log keys (only seeing a log key allowed edge usage). For our Timed workflow inference model, every edge of automata is represented by a tuple of log key and maximum time which can pass between visiting the states connected by this edge. A possible parametrization of this new model is in the length of training sequences used as it is in the original model and estimation of the limited time for each edge where we would suggest a value equal to maximum observed time difference plus standard deviance of all observed time difference.

Algorithm 1 shows training process of Timed workflow inference model, where function $first(s : sequence)$ points to the first element of sequence s and $getChild(pa : parent\ automaton, s : state, e : edge)$ returns successor in automaton pa of state s by edge e. Lines 1 initialize empty automaton for every possible log keys which can appear in training set. Then lines [6-14] iterates over all observations of the sequence without the first one which influences only what automaton will be updated and replicates the sequence into automaton states and edges.

Decision-making algorithm of Time workflow inference drafted in Algorithm 2 iterates over all open execution in lines [4-16] to ensure all can proceed to non-anomalous state with the condition specified in line 10. If this condition is satisfied, all reachable state has to be

Algorithm 1: Training algorithm of Timed workflow inference

```
Input   : S: a set of m sequences in form (log timestamp, log key)
Output: A: set of timed workflow automata
1  init A[x] for ∀x ∈ LK
2  foreach s ∈ S do
3      cs = None
4      pt = first(s).logTimestamp
5      pa = A[first(s)]
6      foreach (lt, lk) ∈ s[1:] do
7          if getChild(pa, cs, lk) does not exists then
8              add a newState to pa as child of cs by edge lk
9          end
10         childState = getChild(pa, cs, lk)
11         timeDiff = lt − pt
12         updateTimedBound(A[first(s)], cs, lk, timeDiff)
13         cs, pt = childState, lt
14     end
15 end
```

added to the current execution for the next decision-making step along with new execution starting from the current log key. Then lines [13-15] close an execution if all states of parent automaton have been visited during that execution. Case of unprecedented log key is covered in line [1-3] with two options: skipping log key which has never been seen within training set or auto labelling such entry as Anomaly. In the algorithm there are several additional functions used: $parent(e : execution)$ returns automaton to which execution e belongs, $visitedAll(e : execution)$ return a set of already visited states during execution e, $bound(pa, s, lk)$ returns a time difference boundary for automaton pa and pair of state and edge (s, lk), $visitedAll(e)$ returns a boolean if set of already visited states by execution e is equal to set of all states in parent automaton of e.

5.2 Timed graph workflow (TGW)

Our graph-based model was heavily inspired by work called Event Regularity and Irregularity in a Time Unit [61]. The original paper builds an oriented graph using each pair of records from training sequences based on their overall appearance in the whole training set (pair has to relatively appear in more than τ sequences). The irregularity scope is then computed based on the shortest path between all pairs of records in an observed samples sequence.

This behaviour demonstrated in the original paper using three same sequences [abc] and threshold parameter τ for adding new edge equal to 2/3. Iterating over all sequences and

Algorithm 2: Decision making of Timed workflow inference

Input : A: set of timed workflow automata,
E: set of running automaton executions,
(lt, lk): tuple representing logged record (log timestamp, log key),
skipUnknown: boolean for decision about unpredented log key
Output: boolean: True - anomaly, False - normal

```
1  if ∄A[lk] then
2  |  if skipUnknown then return False else return True
3  end
4  foreach e ∈ E do
5  |  pa = parent(e)
6  |  if ∃s ∈ visitedStates(e) : ∃getChild(pa, s, lk) and lt < bound(pa, s, lk) then
7  |  |  add to visitedStates(e) ∀x :
8  |  |  ∃s ∈ visitedStates(e) : getChild(pa, s, lk) == x and lt < bound(pa, s, lk)
9  |  else
10 |  |  empty E
11 |  |  return True
12 |  end
13 |  if visitedAll(e) then
14 |  |  E.pop(e)
15 |  end
16 end
17 add empty execution of automaton A[lk] to E
18 return False
```

their pair of records starting from the smallest distance between these records - starting with distance 0 (neighbours), we have 2 potential edges $a \to b$ and $b \to c$, which could be added to the final graph. Both edges appear in all 3 training sequences, it means their appearance is higher than the given threshold and both are added to the final graph. After each stem of such iteration transitive law of edge connection is applied, therefore even edge $a \to c$ is added. From the second iteration over pair which are having one extra record between them we get one potential edge $a \to c$. Its appearance is also higher than the threshold parameter, but since this edge already exists, no action is needed.

We can see this approach as a generalization of workflow inference model, where all workflow automata are merged into a single graph. This allows us to work with sequences which would be labelled as anomalyous by standard Workflow inference model, even though the surrounding context is features to be normal. This model assigns irregularity score to every input test instead of bias labelling. This extra feature copies behaviour of the current state of the art anomaly detection implemented with LSTM neural network (Section 2.2.3.3) which also return only a probability of an event fitting into surrounding context, which has to be experimentally thresholded for anomaly detection.

We also wanted improve irregularity measure using a timing factor ν for record pairs - $(x,\ y)$ - according their timestamp difference - td - and normal distribution over observed differences with respective mean $\mu_{x,y}$ and variance $\sigma_{x,y}$:

$$\nu(x, y, td) = 1 - \frac{1}{\sqrt{2\pi\sigma_{x,y}^2}} e^{-\frac{(td-\mu_{x,y})^2}{2\sigma_{x,y}^2}} \tag{5.1}$$

In our training algorithm we proposed in Algorithm 3 we derive from the original algorithm in [61] using a simple flat structure of events (added value of hierarchies is used in **Timed hierarchical graph workflow**). We also do not utilize threshold parameter τ to limit edges which can be added, since we train on guaranteed normal behaviour sequences. In our implementation, the task is becoming much easier compare to the original one, it is reduces to creating oriented edges for every record pair within a training sequence.

In line 1 we initialize a model Graph, where vertices are all log keys present in training set plus one extra for unknown log key - log key which was not seen during training - and an empty set of edges. Line 2 is the initialization of mean and time difference variance between consequent low pairs. Then in the loop in lines [3-8] we gradually moving execution window of length d and add an edge and update TD statistic features for every pair of the first record in execution and all other records which follow.

Algorithm 3: Training algorithm of Timed graph workflow model

Input : S: a sequences of length m in form (log timestamp, log key)
d: depth of sub-executions - $d << m$

Output: G: workflow graph (V, E)
TD: $(mu_{x,y}, sigma_{x,y})$ for $\forall x, y$ in V

1 G = (V, E) where $V = [\text{Uknown}, unique(S)]$ and E is to be determined
2 TD[x, y] = (0, 0) for $\forall x, y$ in V
3 **foreach** $exec \in S : len(exec) = depth$ **do**
4 | **foreach** (lt, lk) *in* $exes[1:]$ **do**
5 | | $E = E \cup (first(exec), lk)$
6 | | $updateTD(first(exec), lk, lt - first(exec).timeStamp)$
7 | **end**
8 **end**

Following decision-making algorithm in Algorithm 4 which accepts a sequence of records which is shorter than depth parameter d from training algorithm, computer anomaly score based on every oriented pairs of records within the sequence. In lines [4, 6] there is used timing factor ν from (5.1) to discount small constant of existing edge and higher constant of non-existing edge respectively.

Algorithm 4: Decision making of Timed workflow inference

Input : G: workflow graph (V, E)
TD: $(mu_{x,y}, sigma_{x,y}) for\ \forall x, y\ in\ V$
S: S: a sequences of length $m <= d$ in form (log timestamp, log key)
Output: score: anomaly score

1 score = 0
2 **foreach** $(lt_1, lk_1), (lt_2, lk_2) \in S : lt_1 < lt_2$ **do**
3   **if** $(lk_1, lk_2) \in E$ **then**
4   score += $\nu(x, y, td) * 0.1$
5   **else**
6   score += $\nu(x, y, td) * 1$
7   **end**
8 **end**
9 add empty execution of automaton $A[lk]$ to E
10 return False

5.3 Timed hierarchical graph workflow

This model benefits from an additional feature of hierarchy trees from [61]. This would allow us to use log variables as leaves of such tree and log keys as their patterns. Further branching of such tree towards root would be based on log level and log source application as we illustrate in Figure 5.2.

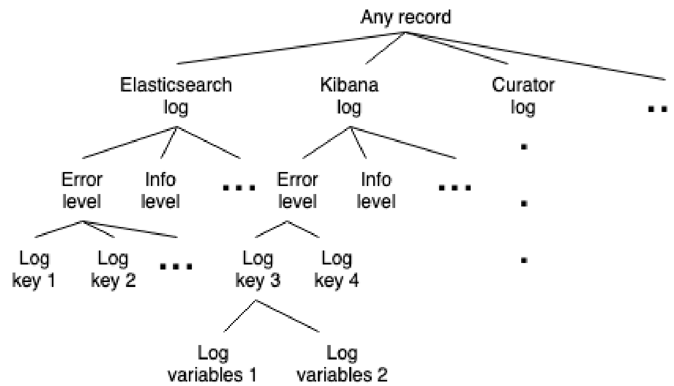


Figure 5.2: Illustration of hierarchy tree for logged records, where dots replace repetitive patters.

The training algorithm is very similar to Algorithm 3 with the difference, every edge propagate to all same-level ancestors of both connected leaves. It means, if there is an edge between Log key 1 and Log key 4 in Figure 5.2, there will also and edge between (Elasticsearch log - Error level) and (Kibana log - Error level) and also between Elasticsearch log and Kibana log.

The intuition behind decision making is based on penalization regularity score of two leaves which do not have a direct connection by amount of level to the root where generalization of these leaves have such connection. For instance if we have just the 3 edges created based on Log key 1 and Log key 4, regularity of incoming Log key 1 and Log key 3 would be penalized by factor 2 because only generalization of these nodes which have a connection are Elasticsearch log and Kibana log.

With this knowledge we can see previous Timed graph workflow as a special case of Timed hierarchical graph workflow with simple hierarchy tree where all leaves are directly connected to root. So the rest of training and decision-making procedure stay unchange.

Chapter 6

Experiments

In this chapter we perform experiments on our ELKR dataset using naive solution, window log key count classification, workflow based model and state of the art LSTM neural network and compare them models we propose in Chapter 5. All the experiments are valuated using standard metrics - precision, recall and f1-scope - to be consistent with other relevant papers. All experiments were written in python and are part of code published with this thesis.

All models used in our experiments were tested on both architectures of ELKR dataset and are trained on all length of training part provided - Section 4.2.2. Then evaluation is based on non-intersecting time windows of length 5 minutes as is the rule we set in Chapter 3. Window is marked by a model as anomalous if the models labelled at least one input within the window time range as anomaly other wise it is labeled as normal execution. The same is applied to windows with true labels containing values: normal, maintenance and anomaly. This two window sequences are then compared according to confusion matrix we set in our problem definition - Table 3.1 and evaluated using precision, recall and f-score. Keep in mind, recall is in this case the most significat castor for us, since we do not want our model to miss any anomalous behaviour.

We are working with 2 naive methods which can serve as a basic base line. The first one - **Error rate** - classifies input sequence as an anomalous if density of error messages within a time range exceeds maximum bound from training set. Then method of **Unseen log keys** labels incoming log keys as an anomaly if that log key has not observed before in training set.

As a representative of window log key count matrix classification model we selected One class SVM, since it has the most highest potential highlighted in recent years. We were experimenting with 3 types of feature representation - plain, simple scaling, IDF-CB weighting (2.3) and the last one proved to be most efficient in combination with standard Radial basis function kernel.

Then we had to implement Workflow inference model, because the is not standardized representation to use and testing what is the right training sequence length. Due to exponential grow of open workflow execution with depth of its automata, we had to limit the length to max 20 records in a sequence. Even in this range we did not observed any improvements in recall and only continues decreasing precision with incremental training sequence length over size 12.

For the LSTM we followed recommendation of [19] as visualized in Figure 2.6 using python library Keras[1]. We stick with 2 layer design and tested some other window sizes around recommended size 10 but not with significant effect on overall performance.

Experiments with Timed workflow inference show longer learning time than original implementation, since there is a lot extra work with computing time bounds, but thanks to earlier closing of workflow execution on the time condition we were more free to test longer training sequences. Unfortunately, even here we did not observe a positive outcome from using training sequences longer than 10. We assume it is a problem of high cardinality of such unique sequences and therefore time bounds cannot be trained properly.

Last but not least, we employed our Timed graph workflow (TGW) which proved to be able to fast train and predicts with high recall value. But also we observed that high length of sequnceces do not yield better performance with recommended values in range $< 6, 15 >$

To compare all the models we selected training and test sequence length equal to 10 and time window for log count matrix of length 5 minutes input parametrs for tested models. This values were selected as a best intersection parameter intesection over model separated test. In Figure 6.1 you can see comparison on Architecture 1 of ELKR dataset and in Fifuge 6.2 the same comparison using Architecture 2.

We can see, naive models lacks behind state-of-the-art solutions which was to be expected since no complex hypothesis was running this models. You can also notice, our models are comparable in Workflow inference model and LSTM in term of Recall, but comes on the second place in term of precision and f-score. This can be explain by the 'timed' aspect of both our models, which send are more likely to predict anomaly to be sure of not missing any.

This behaviour is visible in Table 6.1, which shows another point of view at this problem from perspective of detecting an anomaly as a whole. Every models is say to detect an anomaly as a whole if there was at least one anomaly prediction within the anomaly duration. From this table we can read that LSTM model is able to recognize highers variety of potential anomalous behaviour and our models outperformed standard Workflow inference model.

Even though no anomaly detection model employed was able to reach maximal recall which would be desire, we found the models useful and helpful to daily afford of maintaining

[1] https://keras.io/

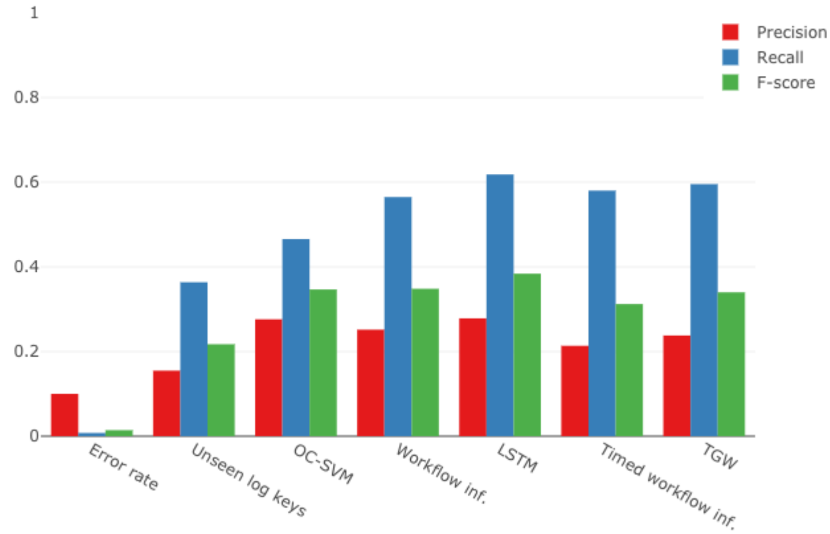


Figure 6.1: Average precision, recall and f-score comparison of studied models on architecture 1 of ELKR dataset

Model	Detected in Arch.1 dataset	Detected in Arch.2 dataset
Error rate	1	1
Unseen log keys	3	1
OC-SVM	3	1
Workflow inference	4	2
LSTM	7	3
Timed workflow inference	5	3
Timed graph workflow	6	3
	anomalies of of 8 in total	anomalies of of 4 in total

Table 6.1: Table of how many continuous anomaly behaviour were detected at least at one point per application and dataset architecture.

complex software architectures. Also it is worth to mention, that resuls we observed from state-of-the-art models do not performer according to expectation in Table 2.5. This goes to the fact, anomaly detection discipline is complex and really case sensitive, we assume, out dataset contains more noise and different scenarios.

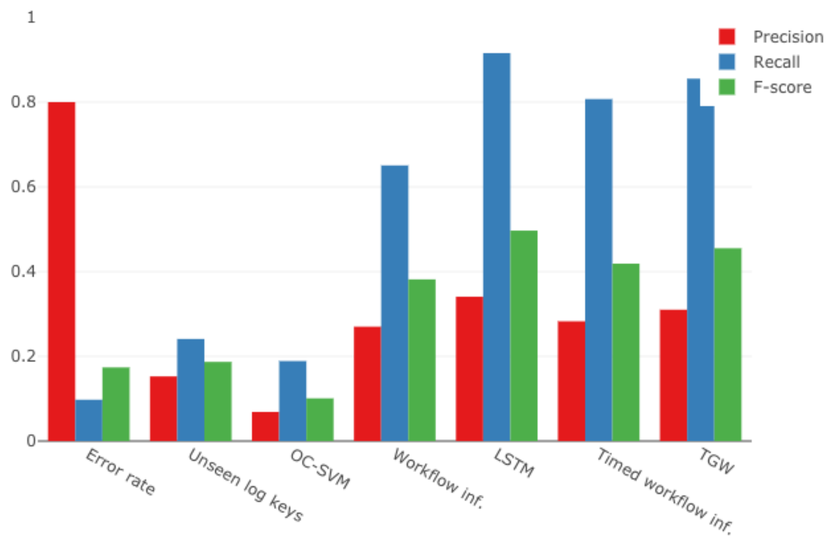


Figure 6.2: Average precision, recall and f-score comparison of studied models on architecture 2 of ELKR dataset

Chapter 7

Conclusion

This research aimed to address a real-world problem of anomaly detection in complex software architectures. To be more specific, we were working with architecture consisting of applications of Elasticsearch platform, RabbitMQ and NGinx, but all the work and model design was done with respect to general rules, structure and consistency to other research works, so it is usable together with any other architecture.

The goal given to us was to propose an automated end-to-end solution, so we had to start with complete state of the art research about possible software and architecture monitoring and behaviour tracking and how this information can be utilized. We were able to build an architecture for log and native metric collection, and such collection could be used for employing a machine learning anomaly detection model. The first challenge was to find a way of processing of all incoming data in form of console logs since every log message is in human readable free text format, which is not suitable as model input. We implemented a source code analysis framework with further log clustering to be able to transform standard log representation into log key structure.

This data was later used in the form of log key count matrix and log key sequence as input for experiments with state of the art anomaly detection algorithm. With further domain analysis, we were able to identify two weak points of workflow-based models. We were able to address the first one by introducing a modification of standard Workflow inference model called **Timed workflow inference** and the second one by a new model based on graph structures called **Timed graph workflow**. Both new models proved to be superior in the field of workflow models and performance-comparable to state of the art of LSTM neural network.

Our solution proved to be an efficient approach of early anomaly detection and in experiments performed with same efficiency as state-of-the-art models. We were able to detect a majority of anomalous behaviour with a limited amount of false positive alerts. But still not

all anomalies were discover by our models neither other state-of-the-art algorithm. Therefore further research is need to find an improvement to current models of propose better information representation and filtering.

The main disadvantage of all anomaly detection models is in the usage of simple log key representation which forgets a big part of collected information. We drafted a solution to this limitation in our **Timed hierarchy graph workflow**, but this model was not included in experiments and model comparison, because we wanted to be comparable with other research work. We plan to use this model in further studies.

Another added value of this work is publishing a research dataset which tracks several days of runtime behaviour and monitoring of two separated architectures with anomaly behaviour labelled by experts, who used their long-range experience with respective architectures. All records are in the form of log key - log variables with rich code support to make it as easy as possible for further research. Additional afford was invested into masking of all potentially sensitive data, so there is no risk of information leak.

The solution design (data collection, processing and evaluation) will be now used by a renown company for daily anomaly detection processes and we were asked to implement it at a higher scale than just one type of architecture as we experimented so far. Therefore we see this research as a big success.

Bibliography

[1] N. R. Adiga, G. Almási, G. S. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, et al. An overview of the bluegene/l supercomputer. In *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 60–60. IEEE, 2002.

[2] S. Ahmad, A. Lavin, S. Purdy, and Z. Agha. Unsupervised real-time anomaly detection for streaming data. *Neurocomputing*, 262:134–147, 2017.

[3] S. Ahmad and S. Purdy. Real-time anomaly detection for streaming analytics. *arXiv preprint arXiv:1607.02480*, 2016.

[4] H. N. Akouemo and R. J. Povinelli. Data improving in time series using arx and ann models. *IEEE Transactions on Power Systems*, 32(5):3352–3359, 2017.

[5] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, (12):33–37, 2007.

[6] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In *Proceedings of the 36th International Conference on Software Engineering*, pages 468–479. ACM, 2014.

[7] R. J. Bolton, D. J. Hand, et al. *Credit Scoring and Credit Control VII*, pages 235–255, 2001.

[8] A. Borghesi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini. Anomaly detection using autoencoders in high performance computing systems. *arXiv preprint arXiv:1811.05269*, 2018.

[9] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM (JACM)*, 30(2):323–342, 1983.

[10] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

[11] B. Chen and Z. M. J. Jiang. Characterizing and detecting anti-patterns in the logging code. In *Proceedings of the 39th International Conference on Software Engineering*, pages 71–81. IEEE Press, 2017.

[12] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *NSDI*, volume 8, pages 337–350, 2008.

[13] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *null*, pages 36–43. IEEE, 2004.

[14] N. Chinchor. Muc-4 evaluation metrics. In *Proceedings of the 4th conference on Message understanding*, pages 22–29. Association for Computational Linguistics, 1992.

[15] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[16] E. M. Dashofy, A. Van der Hoek, and R. N. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, pages 21–26. ACM, 2002.

[17] J. Davis and M. Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240. ACM, 2006.

[18] M. Du and F. Li. Spell: Streaming parsing of system event logs. In *Data Mining (ICDM), 2016 IEEE 16th International Conference on*, pages 859–864. IEEE, 2016.

[19] M. Du, F. Li, G. Zheng, and V. Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298. ACM, 2017.

[20] F. Edgeworth. Xli. on discordant observations. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 23(143):364–375, 1887.

[21] M. Farshchi, J.-G. Schneider, I. Weber, and J. Grundy. Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pages 24–34. IEEE, 2015.

[22] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pages 20–20. USENIX Association, 2007.

[23] K. P. F.R.S. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[24] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 149–158. IEEE, 2009.

[25] S. Fu, J. Liu, and H. Pannu. A hybrid anomaly detection framework in cloud computing using one-class and two-class support vector machines. In *International Conference on Advanced Data Mining and Applications*, pages 726–738. Springer, 2012.

[26] P. Ge. Box., and g. m, jenkins.,"time series analysis, forecasting and control". *ed: San Francisco, CA: Holden Day*, 1970.

[27] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, Bolton Landing, NY, 2003.

[28] N. Goix. How to evaluate the quality of unsupervised anomaly detection algorithms? *arXiv preprint arXiv:1607.01152*, 2016.

[29] C. Gülcü. *The complete log4j manual.* QOS. ch, 2003.

[30] J. Gurland. Hypothesis testing in time series analysis., 1954.

[31] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu. An evaluation study on log parsing and its use in log mining. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 654–661. IEEE, 2016.

[32] P. He, J. Zhu, Z. Zheng, and M. R. Lyu. Drain: An online log parsing approach with fixed depth tree. In *Web Services (ICWS), 2017 IEEE International Conference on*, pages 33–40. IEEE, 2017.

[33] S. He, J. Zhu, P. He, and M. R. Lyu. Experience report: system log analysis for anomaly detection. In *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*, pages 207–218. IEEE, 2016.

[34] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[35] Y.-J. Hong, J. Xue, and M. Thottethodi. Dynamic server provisioning to minimize cost in an iaas cloud. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 147–148. ACM, 2011.

[36] W. Hu, Y. Liao, and V. R. Vemuri. Robust anomaly detection using support vector machines. In *Proceedings of the international conference on machine learning*, pages 282–289, 2003.

[37] P. Joshi, H. Zhang, X. Jianwu, X. Yu, and G. Jiang. Cloudseer: using logs to detect errors in the cloud infrastructure, Aug. 1 2017. US Patent 9,720,753.

[38] A. S. Kumar and S. Mazumdar. Forecasting hpc workload using arma models and ssa. In *Information Technology (ICIT), 2016 International Conference on*, pages 294–297. IEEE, 2016.

[39] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo. Failure prediction in ibm bluegene/l event logs. In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*, pages 583–588. IEEE, 2007.

[40] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen. Log clustering based problem identification for online service systems. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 102–111. ACM, 2016.

[41] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li. Mining invariants from console logs for system problem detection. In *USENIX Annual Technical Conference*, pages 23–25, 2010.

[42] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley, Calif., 1967. University of California Press.

[43] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1255–1264. ACM, 2009.

[44] H. Mi, H. Wang, Y. Zhou, M. R.-T. Lyu, and H. Cai. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1245–1255, 2013.

[45] M. Mizutani. Incremental mining of system log format. In *Services Computing (SCC), 2013 IEEE International Conference on*, pages 595–602. IEEE, 2013.

[46] M.-N. Nguyen and N. A. Vien. Scalable and interpretable one-class svms with deep learning and random fourier features. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 157–172. Springer, 2018.

[47] A. J. Oliner, A. Aiken, and J. Stearley. Alert detection in system logs. In *2008 Eighth IEEE International Conference on Data Mining*, pages 959–964. IEEE, 2008.

[48] A. Rabkin and R. Katz. Chukwa: A system for reliable large-scale log collection. In *Proceedings of LISA'10: 24th Large Installation System Administration Conference*, page 163, 2010.

[49] M. Rehák, M. Pěchouček, M. Grill, and K. Bartos. Trust-based classifier combination for network anomaly detection. *Cooperative Information Agents XII*, pages 116–130, 2008.

[50] L. Rettig, M. Khayati, P. Cudré-Mauroux, and M. Piorkowski. Online anomaly detection over big data streams. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 1113–1122. IEEE, 2015.

[51] T. Saito and M. Rehmsmeier. The precision-recall plot is more informative than the roc plot when evaluating binary classifiers on imbalanced datasets. *PloS one*, 10(3):e0118432, 2015.

[52] O. Salem, Y. Liu, A. Mehaoua, and R. Boutaba. Online anomaly detection in wireless body area networks for reliable healthcare monitoring. *IEEE journal of biomedical and health informatics*, 18(5):1541–1551, 2014.

[53] S. L. Salzberg. C4. 5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993. *Machine Learning*, 16(3):235–240, 1994.

[54] B. Schölkopf, R. C. Williamson, A. J. Smola, J. Shawe-Taylor, and J. C. Platt. Support vector method for novelty detection. In *Advances in neural information processing systems*, pages 582–588, 2000.

[55] C. Scott. Performance measures for neyman–pearson classification. *IEEE Transactions on Information Theory*, 53(8):2852–2863, 2007.

[56] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. Ieee, 2010.

[57] L. Tang, T. Li, and C.-S. Perng. Logsig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 785–794. ACM, 2011.

[58] A. Tomé, D. Malafaia, A. Teixeira, and E. Lang. On the use of singular spectrum analysis. *arXiv preprint arXiv:1807.10679*, 2018.

[59] H. Trevor, T. Robert, and F. JH. The elements of statistical learning: data mining, inference, and prediction, 2009.

[60] R. Vaarandi. A data clustering algorithm for mining patterns from event logs. In *IP Operations & Management, 2003.(IPOM 2003). 3rd IEEE Workshop on*, pages 119–126. IEEE, 2003.

[61] L. Wan and T. Ge. Event regularity and irregularity in a time unit. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 930–941. IEEE, 2016.

[62] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Online system problem detection by mining patterns of console logs. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 588–597. IEEE, 2009.

[63] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132. ACM, 2009.

[64] J. Zhu. HDFS Logs. 7 2017.