

# Monitoring Release Logs at Adyen: Feature Extraction and Anomaly Detection

---

*Version of August 23, 2018*

Yikai Lan



---

# Monitoring Release Logs at Adyen: Feature Extraction and Anomaly Detection

---

THESIS

submitted in partial fulfillment of the  
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Yikai Lan  
born in Hunan, China



Software Engineering Research Group  
Department of Software Technology  
Faculty EEMCS, Delft University of Technology  
Delft, the Netherlands  
[www.ewi.tudelft.nl](http://www.ewi.tudelft.nl)



Adyen  
Simon Carmiggeltstraat 6-50, 1011DJ  
Amsterdam, the Netherlands  
[www.adyen.com](http://www.adyen.com)



---

# Monitoring Release Logs at Adyen: Feature Extraction and Anomaly Detection

---

Author: Yikai Lan  
Student id: 4474783  
Email: yikailan.l@gmail.com

## Abstract

Monitoring the release logs of modern online software is a challenging topic because of the enormous amount of release logs and the complicated release process. The goal of this thesis is to develop a pipeline that can monitor the release logs and find anomalous logs, automating this step with anomaly detection and reducing the required manual effort. We improve the pipeline from the recent work of Microsoft [34], enabling it to monitor logs with different severity levels and extremely long sequences.

We first use IPLoM and its reconciling step for raw logs to obtain log events and then use log event sets, a simplified version of log sequences, for anomaly detection. The outlier scores of log event sets are calculated using anomaly detection algorithms, and those with an outlier score higher than the threshold are clustered to reduce the number of output. In the final output result, we propose two ranking functions to sort the potential anomalous clusters and only show the top 10 results. Another complementary step beside anomaly detection is designed to capture recurrent anomalies in known clusters that have seen before. By finding the optimal parameters for hierarchical clustering, nearest neighbor distance, and LOF, we test the performance of pipeline on Adyen log data and make our suggestions. Finally, we also test the robustness of the pipeline with two types of artificial data sets.

## Thesis Committee:

Chair and supervisor:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. Ir. S.E. Verwer, Faculty EEMCS, TU Delft
Company supervisor:	Ir. P.F. Huibers, Adyen B.V.
Committee Member:	Dr. D.M.J. Tax, Faculty EEMCS, TU Delft



---

# Preface

Finally, it is my turn to write the preface that expresses my gratitude to the ones that help me during my thesis and my study. It has been the most challenging and exciting years in my life to study computer science at TU Delft.

My master thesis so far is my favorite project: it focuses on an interesting topic that applying anomaly detection techniques for software log analysis, providing possibilities for the future. I want first to thank Bert Wolters and Joop Aue for providing me the valuable opportunity to do an internship in Adyen in the first place. Adyen is made up of a group of enthusiastic people that I can learn a lot from. I want to thank Pieter Huibers, my company supervisor, for providing the necessary resources and suggestions from the developers' side which are critical to accomplishing this project. Also, thank you for your support during my internship when I was trying to find the direction of the project.

Moreover, I would like to thank Prof. Arie van Deursen for giving valuable opinions for my project and spending time reading my thesis. Your idea helps me a lot in shaping the outline of the thesis. I want to also thank Mauricio for providing great ideas and guidance for my project when I was struggling with the feature extraction. Thank you for your encouragement when I started to doubt the purpose of this project.

Sicco, my supervisor of the thesis, has helped a lot in shaping my thoughts to propose the research questions. Thank you for guiding me through the experiments when I am confused. Your suggestions on the overall structure and spending the time to read my writing improve the quality of this thesis.

I want to thank my colleagues Alexandru, Daan, Diana, Joop, Jos, Peter, Salvatore, Serge, Traian and Warwick for giving me suggestions on my project. I would also want to thank Cristian, Daniel, and Jelle, who sit next to me, for making the work so much fun. I would also like to thank my friends Agathe, Shiwei, Yadong, and Zina to discuss with me about this project. Finally, I want to thank my family for their unconditional support and confidence in me. I am really grateful for those who help me during my study and my internship in Adyen. I also hope that my work in this project can help people who want to solve the same problem and provide an interesting starting point for future work.

Yikai Lan  
Delft, the Netherlands

August 23, 2018



---

# Contents

<b>Preface</b>	<b>iii</b>
<b>Contents</b>	<b>v</b>
<b>List of Figures</b>	<b>vii</b>
<b>List of Tables</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Deployment and Release . . . . .	1
1.2 Logs for Monitoring Release . . . . .	2
1.3 Log Abstraction Methods . . . . .	3
1.4 Log Event Sequences and Log Event Sets . . . . .	4
1.5 Weighting Log Events . . . . .	4
1.6 Anomaly Detection Using Log Event Sets . . . . .	4
1.7 An Example Data Set for the Pipeline . . . . .	5
1.8 Adyen and Its Deployment Strategy . . . . .	7
1.9 Research Questions . . . . .	9
1.10 Structure of the Thesis . . . . .	10
<b>2 Related Work</b>	<b>11</b>
2.1 Previous Log Analysis Work in Adyen . . . . .	11
2.2 Log Abstraction Methods . . . . .	11
2.3 Anomaly Detection Algorithms . . . . .	14
<b>3 Feature Extraction</b>	<b>19</b>
3.1 Data: Logs in Adyen . . . . .	19
3.2 Parsing Logs or Not? . . . . .	21
3.3 After Parsing: Extracting Features from Release Logs . . . . .	23
3.4 Definitions and Assumptions in Anomaly Detection . . . . .	28

<b>4</b>	<b>Pipeline of Monitoring Release Logs</b>	<b>31</b>
4.1	Collecting Log Data . . . . .	33
4.2	Parsing Logs to Templates . . . . .	33
4.3	From Log Sequence to Binary Feature Vector . . . . .	38
4.4	Weighted Feature Vector . . . . .	41
4.5	Distance and Similarity . . . . .	43
4.6	Anomaly Detection . . . . .	43
4.7	Final Output of Potential Anomalous Clusters and Known Clusters . . . . .	47
<b>5</b>	<b>Experiment Results</b>	<b>49</b>
5.1	Experiment Setup . . . . .	49
5.2	Feature Extraction Results . . . . .	57
5.3	Anomaly Detection Algorithms . . . . .	59
5.4	Final Test Result . . . . .	79
<b>6</b>	<b>Experiment Thoughts</b>	<b>81</b>
6.1	Data Set . . . . .	81
6.2	Extracting Features from Logs . . . . .	82
6.3	Anomaly Detection . . . . .	83
6.4	Log Practice . . . . .	84
<b>7</b>	<b>Conclusions and Future Work</b>	<b>85</b>
7.1	Contributions . . . . .	85
7.2	Limitations . . . . .	86
7.3	Conclusions . . . . .	88
7.4	Future work . . . . .	88
	<b>Bibliography</b>	<b>91</b>

---

## List of Figures

1.1	A deployment pipeline example. . . . .	2
1.2	A log example from [22] . . . . .	3
1.3	Constructing weighted vectors from log event sequences in log data. . . . .	6
1.4	Code releases in Adyen. . . . .	8
1.5	A Kibana dashboard example from Elasticsearch official website. . . . .	8
3.1	A log detail from <i>LogSearch</i> in Adyen. . . . .	20
3.2	A heat map of the correlation matrix of occurrence number of logs from different classes in Adyen. . . . .	22
3.3	The frequency of INFO and WARN logs per second within one host in a release. . . . .	24
4.1	Pipeline of Monitoring Release Logs. . . . .	32
4.2	An input example of concatenation of message and extra attributes for the message abstracting method IPLoM (Iterative Partitioning Log Mining). . . . .	34
4.3	An example of anonymizing pure digits parameters in log messages. . . . .	35
4.4	An example of extracting keys from JSON format logs. . . . .	35
4.5	An example of log message abstraction. . . . .	36
4.6	An example of template clustering. . . . .	39
5.1	The distribution normalized <i>idf</i> weight of log templates from Release Instance 1. . . . .	58
5.2	The distribution of overall weights of log templates from Release Instance 1 . . . . .	58
5.3	The distributions of length and time duration of log templates weights from Release Instance 1. . . . .	60
5.4	The average effort reduction of releases in the training set with different distance metrics under different cluster percentiles. . . . .	62
5.5	The distribution of outlier scores of Release Instance 1 computed by hierarchical clustering (Equation 4.10). . . . .	63
5.6	Boxplot of outlier scores of anomalous event sets in each release instance from the training set, which is computed by hierarchical clustering in different metrics. Distance values of three distance metrics are separated into three groups. . . . .	64

## LIST OF FIGURES

---

5.7	Boxplot of percentile values of outlier scores of new log data in each release instance form the training set, which is computed by hierarchical clustering in different metrics. . . . .	65
5.8	MAP and average recall of clustering for original data set in the training set using different <i>anomaly cluster distances</i> . . . . .	66
5.9	MAP and average recall of clustering for INFO anomalies data set using different <i>anomaly cluster distances</i> . . . . .	66
5.10	MAP and average recall of clustering for original data set in the training set using different <i>first cluster distances</i> . . . . .	68
5.11	MAP and average recall of clustering with INFO anomalies data set in the training set using different <i>first cluster distances</i> . . . . .	68
5.12	Ratio of history cluster number and history log event set using different <i>first cluster distances</i> in the training set. . . . .	69
5.13	MAP and average recall of clustering for original data set using different <i>severity ratios</i> . . . . .	70
5.14	MAP and average recall of clustering for INFO anomalies data set using different <i>severity ratios</i> . . . . .	70
5.15	MAP and average recall of nearest neighbor distance for original data set using different $k$ values (severity ratio = 6). . . . .	73
5.16	MAP and average recall of nearest neighbor distance for INFO anomalies under different $k$ values (severity ratio = 6). . . . .	73
5.17	Distance of anomalous event sets with different $k$ values. . . . .	74
5.18	Distance of new log event sets with different $k$ . . . . .	74
5.19	MAP and average recall of nearest neighbor algorithm for original data set with different severity ratios ( $k = 5$ ). . . . .	75
5.20	MAP and average recall of nearest neighbor to capture INFO Anomalies under different severity ratios ( $k = 5$ ). . . . .	76
5.21	MAP and average recall of LOF for original data set under different $k$ (severity ratio = 4). . . . .	77
5.22	MAP and average recall of LOF for INFO anomalies data set under different $k$ (severity ratio = 4). . . . .	77
5.23	MAP and average recall of LOF for original data set using different severity ratios ( $k = 15$ ). . . . .	78
5.24	MAP and average recall of LOF for INFO Anomalies using different severity ratios ( $k = 15$ ). . . . .	78

---

## List of Tables

1.1	2-nearest neighbor average distance on example data set . . . . .	7
5.1	Release instance information in training set . . . . .	51
5.2	Release instance information in test set . . . . .	52
5.3	Average recall and average overall recall of pipeline on recurrent anomalies data set. . . . .	72
5.4	Optimal Parameters for Anomaly Detection Algorithms. . . . .	79
5.5	Test Set Results. . . . .	80
7.1	The number of new log events in release instances. . . . .	87



# Chapter 1

---

## Introduction

Monitoring the release of modern online software is a quite challenging topic as there may be hundreds or even thousands of developers working on one software. Changes to the code and configuration may work in the test environment, but anything could go wrong in the production environment. The release is generally a manually intensive process, which requires experienced developers who have the expertise of the whole software to monitor the process. The goal of this thesis is to develop a pipeline that can monitor the release using logs as input, automating this step and reducing the required manual effort.

The log data set in this thesis is provided by Adyen, a financial-technology company based in Amsterdam. Some of the hosts in Adyen can generate millions of log entries with different verbosity levels from various applications during a release. For this thesis, we develop a pipeline that utilize anomaly detection to capture logs that may cause problems for the release and compare it with recent research from Microsoft [34].

Some fundamental background concepts within the thesis, such as deployment, release, logs and anomaly detection, are first introduced in this chapter. Research questions are then proposed as the guidelines of the thesis. Finally, the structure of the thesis is stated briefly in the last section.

### 1.1 Deployment and Release

The deployment of software is a repeated process where new software is introduced to the infrastructure to address the business need [49]. The release of software is the final step of bringing the code changes to the end users [1] and is the step when the code is updated, compiled, and launched on the hosts for online systems. In other words, the release is an important step in the deployment process.

A deployment pipeline is used to describe the deployment process, from software development to the release. An example of a designed pipeline is shown in Figure 1.1 from [26]. The deployment process consists of building, deploying, testing and releasing software [26]. Being the first step of the pipeline, the commit stage in the leftmost makes sure that the code meets the lowest standard of requirements, which consists of successfully compiling the code, passing the automatic unit tests, and running code analysis. The second

## 1. INTRODUCTION

---

step is the automatic acceptance testing that checks if the system meets basic needs of users or functions. Later, automatic capacity testing is used to model the production environment as much as possible to make sure the system is of high performance. The fourth step in the pipeline is the manual testing which runs to capture errors that are not captured by the automatic tests. The release is the last step and might be the scariest step for most developers because any error within the pipeline would likely lead to failure of the release step. This includes run-time errors, performance issues and even failure of the systems. As suggested in [26], the deployment of the software and configuration management should all be done automatically to reduce the errors in the release in the production environment.

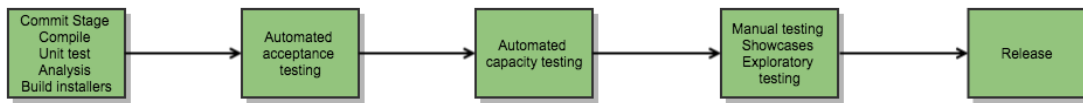


Figure 1.1: A deployment pipeline example from [26]. The first step is the commit stage where code is compiled and goes through basic unit tests. The second step is the automatic acceptance testing for basic functions, after which software would be tested for capacity using data similar to production environment. Later manual testing in the fourth step can further capture errors neglected by automatic tests. The release is the final step of the pipeline, bringing the software to production to the end users.

## 1.2 Logs for Monitoring Release

Software logs, where status and actions of the software are logged into, are a useful resource for developers to understand the behavior of software and thus play an essential role in monitoring the status of the hosts for companies with online services systems. Logs mainly consist of plain text attributes that are written by developers to record information using different logging formats.

Besides being used in daily monitoring, logs can also be used for monitoring code release. Release monitoring is typically a manual step, which can be time-consuming and tedious because of the need to inspect logs and the statistic data of the hosts, such as connection timeout frequency and warnings number. Also, a manual procedure has to be repeated for each deployment and does not scale well when the number of hosts in the system continues to increase.

As shown in Figure 1.1, most tests and steps including the release step should be automatic for the scalable growth of the business. Though different companies may have different deployment pipelines, deployment should follow the automatic patterns as much as possible. Manually monitoring release should be replaced by automatic monitoring to achieve a more smooth and less risky deployment as stated by [26].



### 1.3 Log Abstraction Methods

Given the massive log data generated during the release of the production environment, it is best to use anomaly detection techniques to automate the release monitoring. However, the logs need to be preprocessed into numerical feature vectors from plain text attributes before being used as the input of anomaly detection algorithms. Rarely two logs would be exactly the same because logs also contain variable information that is changed whenever a log is generated. Thus, analyzing raw logs directly is barely possible and may lead to curse of dimensionality. To solve this problem, a log abstraction method is used to parse the raw log messages and generate log templates. Each template is a plain text that is shared by multiple logs and represents an event or status of the system after filtering out irrelevant information. Compared to the actual number of raw logs, the number of templates is much smaller since it would be very close to the actual number of code lines that generate logs in an ideal situation.

```
2008-11-09 20:35:32,146 INFO
Receiving block blk_-1608999687919862906 src: /10
.251.31.5:42506 dest: /10.251.31.5:50010
```

Figure 1.2: A log example from [22]. “2008-11-09 20:35:32,146” is the timestamp that the log is generated. “INFO” is the verbosity level of the log, indicating the severity level. The text in light grey background is the log message. In the log message, constant parts are of blue font color and variable parts of black font color.

Raw logs stored in hosts usually consists of multiple attributes, including timestamp, verbosity level, and log message. In the log example in Figure 1.2 [22], “2008-11-09 20:35:32,146” is the timestamp that the log has been generated and “INFO” is the verbosity level or severity level. The log message is the text in the grey background, which consists of two parts: *constant parts* of blue font color and *variable parts* of black font color. Constant parts are originally written by developers and are shared by multiple logs. Variable parts carry run-time information such as the system status or a task ID. In this message, we have two types of variable parts block ID and IP address. “blk\_-1608999687919862906” is the block ID in Hadoop File System. “/10.251.31.5:42506” and “/10.251.31.5:50010” are IP addresses. The raw log message should be parsed into a log event as “Receiving block \* src: \* dest: \*” by replacing the variable parts including block ID, source and destination IP address in the raw message.

Based on whether to use source code or not, log message abstraction methods can be divided into two categories. If the source code is available, it is possible to find the code line that generates logs depending on the language and generate log events directly from the source code, for example by finding *print* statements. If the source code is not available, log events can also be found by analyzing solely the raw logs and still achieve a high accuracy [22]. One of the common strategies of such method with only raw logs as input is to use

heuristic rules, such as assuming that tokens in front may have a higher chance to be constant and pure digits may be more likely to be variable parts [27][28][23]. The other common strategy is to use a clustering algorithm suitable for raw logs [19][36][37][50].

More details about the log abstraction methods can be found in Chapter 2.

### 1.4 Log Event Sequences and Log Event Sets

A task in software usually refers to a job to achieve a particular purpose, such as one payment task in Adyen or a Map-Reduce job in Hadoop. Adyen uses two type of task IDs, PSP references and thread IDs, to track one task in the systems. PSP references are the unique task IDs generated for tasks such as payment processes and are shared by logs among different hosts. Thread IDs are more general task IDs if PSP references are missing. Other software can have different task IDs. The log with same task IDs can form into a log event sequence, which represents a specific step that the logs go through. When a task goes through an entirely different log event sequence, this may be an unwanted error in the system.

To further simplify the log event sequences, the permutations and order of the log events are ignored. In other words, only the occurrence of the log events is considered. All log event sequences would become **log event sets**. If two log event sequences have the same combination of log events, they would end up with the same log event sets.

### 1.5 Weighting Log Events

Different events have different significance in identifying the potential problems in a release, for example, an ERROR event is more likely to be a bug in most of the case than an INFO event. Thus, three intuitive weights are proposed in the pipeline.

1. **Frequency weight**: the less frequent events indicate more important information.
2. **Contrast-based weight**: events which never occur before release should be more important than those which have showed up before.
3. **Severity weight**: events with higher severity have more discriminative power than those with lower severity.

After calculating each weight for one event, the three weights can be combined as shown in Equation 4.7. More details about the weighting can be found in Section 4.4.

### 1.6 Anomaly Detection Using Log Event Sets

After using log abstraction methods to parse the raw logs and get log event sets features, an anomaly detection algorithm needs to monitor the release using the log events. Anomaly detection is a technique that finds patterns in normal data and captures data points that behave differently from the expectation [10]. The goal of anomaly detection is to build

a model that represents the normal data points well and to label any point that deviates far away from normality an anomaly. Thus the key points are how to build the model for normal log data and how to measure the deviation of the new log data points from the normal log data.

Anomaly detection can be categorized as classification based, clustering based, nearest neighbor based, statistical based, information theoretic based, and spectral based [10]. It has broad applications in industries, such as intrusion detection systems, credit-card fraud, medical diagnosis, and law enforcement [2].

New log events are generated because of the update of the source code, which means new features are created after release. It is hard to determine which log events are anomalies based merely on the severity. Extracting features from logs is an essential challenging part in the pipeline. Besides, the chosen anomaly detection algorithm should be able to give an interpretable result and provide feedback to the developers. More details on anomaly detection algorithms using logs as input can be found in Chapter 2.

## 1.7 An Example Data Set for the Pipeline

In this section, we use an example data set from [34] as the input of the designed pipeline. Only some essential steps are included in this section to give a rough introduction. Other detailed designs of pipeline can be found in Chapter 4.

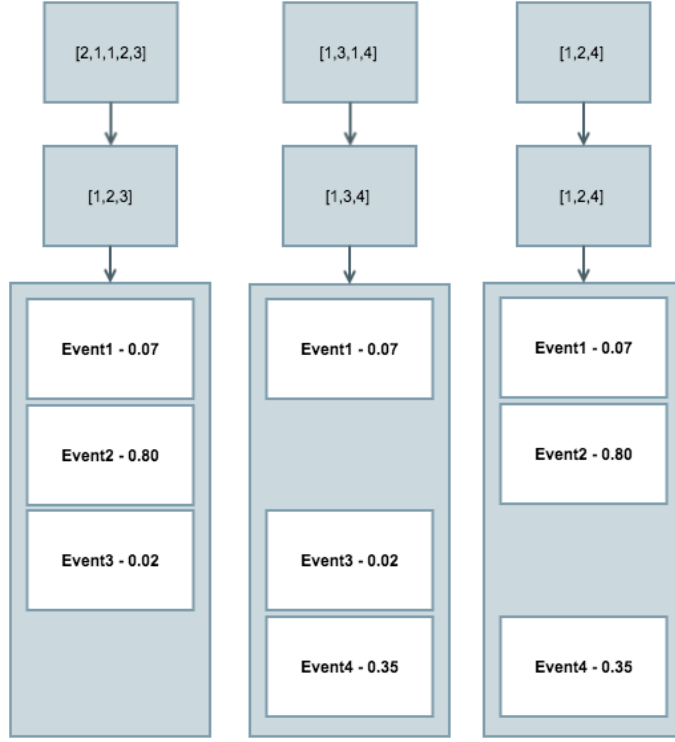
First, we use a log abstraction method to map every log to a log template by its message attribute. Logs with the same task ID would be linked together to build a log sequence. Later, we divide log sequences into history log data and new log data based on the timestamp. We would use log event sets in history log data to detect anomalies in new log data. At last, nearest neighbor distance is used as the anomaly detection algorithm to calculate outlier scores for new log data.

### 1.7.1 From Log Event Sequences to Weighted Vectors

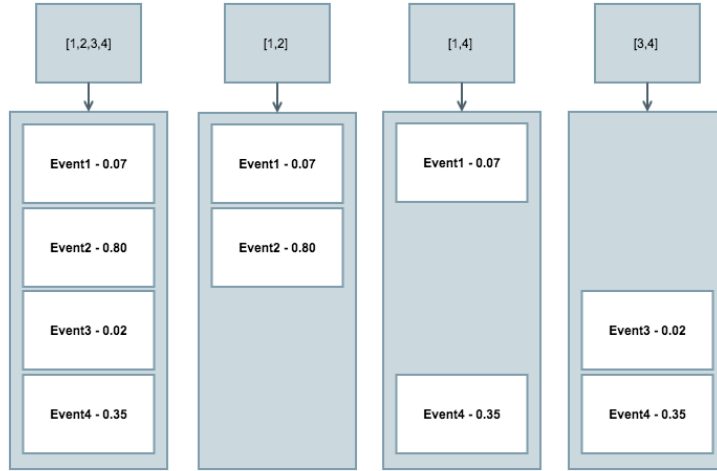
After applying a log abstraction method to the raw logs and tracking the task ID to form log sequences, we obtain three log sequences from our data set:  $[2, 1, 1, 2, 3]$ ,  $[1, 3, 1, 4]$  and  $[1, 2, 4]$ . The digits in the sequences represent the event IDs obtained from the log abstraction method. The order of the digits in log sequences represent the relative order within the sequences according to timestamp. Taking  $[1, 3, 1, 4]$  as an example and assuming its task ID is 2, the task ID 2 goes through event 1, then event 3, later again event 1 and finally event 4.

As shown in Figure 1.3a [34], three log sequences are first processed into log event sets by removing all duplications and ignoring the internal order of events. For example, we first neglect the original orders in the left sequence  $[2, 1, 1, 2, 3]$  from Figure 1.3a and sort it in the event IDs, the sequence then turns into  $[1, 1, 2, 2, 3]$ . Later, the duplicates are again removed, finally a log event set  $[1, 2, 3]$  is obtained.

Because there are in total 4 types of log events in our data set, each log event set is converted into a 4-dimensional feature vector.



(a) Constructing weighted vectors from log event sequences in history log data. The uppermost part consists of the original log event sequences formed by tracking the task IDs. The middle rectangles are log event sets derived from the log event sequences by removing duplications and permutations. The last part consists of weighted vectors where the weights are calculated based on Section 1.5.



(b) Constructing weighted vectors from log event sequences in new log data. Weights of log events are from [34]. Log event sequences are not shown in this example. The uppermost part consists of log event sets derived from the log event sequences by removing duplications and permutations. The bottom part consists of weighted vectors where the weights are calculated based on Section 1.5.

Figure 1.3: Constructing weighted vectors from log event sequences in history log data and new log data [34].

### 1.7.2 Anomaly Detection Using Nearest Neighbor Distance

After obtaining the 4-dimensional vectors in Figure 1.3a, anomaly detection algorithms can be used to calculate the outlier scores for new log sequences. The history log data consists of three log event sets:  $[1, 2, 3]$ ,  $[1, 3, 4]$  and  $[1, 2, 4]$  from Figure 1.3a. We assume that new log data consists of  $[1, 2, 3, 4]$ ,  $[1, 2]$ ,  $[1, 4]$  and  $[3, 4]$  as shown in Figure 1.3b [34]. We use 2-nearest neighbor average distance as an outlier score for each new log event set. There are also other anomaly detection algorithms to calculate outlier scores as shown in Section 4.6.

For each new log event set, we find the 2 nearest neighbors in the history log data, and calculate the average distance between the new log data and the 2 nearest neighbors. The result is shown in Table 1.1.  $[1, 4]$  and  $[3, 4]$  have a higher outlier score of 0.410 and 0.436. Thus they would be more likely to be anomalies in this data set.

Table 1.1: 2-nearest neighbor average distance on example data set

New Log Event Sets	2-Nearest Neighbor Average Distance
$[1, 2, 3, 4]$	0.185
$[1, 2]$	0.185
$[1, 4]$	0.410
$[3, 4]$	0.436

## 1.8 Adyen and Its Deployment Strategy

In this thesis, research is conducted using log data from Adyen. Before building the pipeline, it is important to understand the deployment strategy of Adyen and how developers in Adyen monitor release.

### 1.8.1 About Adyen

Our industry partner Adyen is a fast growing financial technology company. It aims at providing merchants with one single payment platform solution that allows them to do business around the globe. Since Adyen is growing fast, the code is frequently updated which requires scalable release monitoring during the deployment.

### 1.8.2 Deployment Strategy and Using Logs for Debug

There are three types of code releases each week in Adyen which are in the order of *Beta*, *Test* and *Live* as shown in Figure 1.4. Depending on the type of release, Beta Hosts, Test Hosts or Live Hosts would go through the release processes at each release day separately. Beta Hosts are internal hosts where artificially generated payments and other processes are used to test for any deviations from the expected behavior with the new code version. Test Hosts are also used to test new functions but with real processes. Live Hosts are the hosts in the production environment.

## 1. INTRODUCTION

---



Figure 1.4: Code releases in Adyen. The first release is the Beta release where new functions are tested with artificial data. The second release is the Test release, allowing part of the users actually test the new functions. The Live release is the last one, where all users can get access to the new functions.

In this thesis, only log data collected in the live environment is used for analysis and experiments. At Adyen, an enormous amount, currently over 1 billion, of logs are generated every day using a Java logging framework Log4j<sup>1</sup>. A detailed log entry example is shown in Figure 3.1. The logs from different hosts are stored in an Elasticsearch database and visualized using Kibana dashboard as shown in Figure 1.5<sup>2</sup>. Elasticsearch is a distributed search engine which provides fast search on JSON-like documents [14]. Elasticsearch is convenient for searching logs as users can filter values in certain keys.



Figure 1.5: A Kibana dashboard example from Elasticsearch official website. The histograms show the frequency of a certain searching query. The details of each log entry are not shown in this example, but they are usually at the bottom of the histograms.

Adyen uses the green-blue deployment pattern to manage releases. In the green-blue deployment pattern, we keep two same production environments; one is called the blue environment, and the other one is called the green environment [26]. For example, if the current release fails in the blue environment, we can use the load-balancer to switch back to the green environment which is still running code from the previous release. During the release process, a newly released host is compared with other hosts running the previous

---

<sup>1</sup><https://logging.apache.org/log4j/2.x/>

<sup>2</sup><https://www.elastic.co/guide/en/beats/metricbeat/current/view-kibana-dashboards.html>

release in the platform. Release monitoring has manual involvement of developers and operations staff. Developers and operations staff are responsible for monitoring logs of the current host and the performance of the whole platform. They would investigate for signs of new issues or newly introduced bugs of the platform. This monitoring method requires knowledge of the whole platform.

Adyen uses an inhouse developed tool to monitor individual host during a release. This is a rule-based method that requires expertise of the code and system. A web interface tool named *Logness* [17], which collects all history WARN and ERROR logs and clusters them together according to the similarity of the message strings, are also used to monitor release by sending a notification about new log clusters. More details about these tools can be found in Section 2.1.

## 1.9 Research Questions

The goal of this research is to develop a log monitor pipeline that captures the most problematic logs during a release. Those may be the cause of problems such as failures or performance issues. Based on the previous works and current monitoring method of Adyen, we propose three research questions to help develop a useful and robust pipeline using anomaly detection.

Since the most challenging part of the pipeline is to model the log events after applying a log abstraction method, our first question is:

*RQ1: Is it useful to apply the weighted vector model to log event sets in monitoring release logs of Adyen by anomaly detection?*

After modeling log events, an anomaly detection algorithm needs to be chosen to discover the log events that may lead to problems in the host. Based on simplicity and interpretability, three proximity based algorithms are tested in an experiment with corresponding optimal parameters. Apart from these parameters, a distance metric needs to be chosen to define the proximity or similarity of different data points. As most of the data points are quite sparse compared to the number of features, the choice of distance metrics affects the performance of the anomaly detection algorithms greatly. Our second research question is:

*RQ2: What are the optimal parameters and distance metrics for proximity-based anomaly detection algorithms that are used in release log monitoring?*

As shown in Section 1.5, the basic assumptions of anomaly detection are that events with lower frequencies, higher severities and newly seen in the release have a higher chance to be anomalies. After all the optimal distance metrics and corresponding parameters are found for anomaly detection algorithms, the anomaly detection algorithms need to be tested with the cases when one of the assumptions of anomalies does not hold.

To this end, two additional artificial data sets are generated with the original data set separately and used to see if the pipeline is robust enough. The first artificial data set is

*INFO anomalies data* set where all anomalous log events are all set to INFO severity. The other artificial data set is *recurrent old anomalies data set*, where the start time of the release is changed in such way to make the anomalies occur for the first time just before the release and happen again during release or after release. With these two data sets, the robustness of the pipeline can be tested. The final research question is:

*RQ3: Is the anomaly detection pipeline robust in the release when assumptions of the anomaly do not hold?*

### 1.10 Structure of the Thesis

The structure of the thesis is listed as follows: Related work is discussed in Chapter 2. In Chapter 3 we explore log patterns in release and discuss how to extract features from release logs. The whole pipeline is stated in Chapter 4. Experiment results regarding different metrics and parameters are shown in Chapter 5. Later some reflections on the project itself are stated in Chapter 6. In Chapter 7, conclusions and potential future work are discussed.



## Chapter 2

---

# Related Work

Quite a few works have used different anomaly detection algorithms on log analysis. In order to understand how anomaly detection can help with monitoring release logs, recent works of related topics, such as log abstraction and anomaly detection of logs, are described in this chapter. Firstly, previous studies in Adyen focusing on log analysis are introduced in Section 2.1. In Section 2.2, we discuss some message abstracting methods that parse raw logs and reduce the number of logs for analysis. Finally, log anomaly detection algorithms are presented in Section 2.3.

### 2.1 Previous Log Analysis Work in Adyen

Adyen already conducts a few research that focuses on log analysis. Peter Evers [17] builds a tool called *Logness* with a web interface, which utilizes *Longest Common Subsequence* algorithm to cluster logs with a severity level of WARN and ERROR together. Logness would send a notification to developers if a new log cluster is created, which can be used to monitor release as well. Rick Wieman [51] evaluates different passive learning methods and uses them to detect anomalies in Point of Sale transaction. Joop Aué [5] researches WEB API usage in Adyen and recommends ways of avoiding API faults.

All these works provide an insight into logs with Adyen from a different perspective and provide helpful ideas for this thesis project.

### 2.2 Log Abstraction Methods

Reducing the size and dimension of input data is crucial to analyze log efficiently and accurately when dealing with a large amount of log data. Logs consist of two different parts, commonly-shared *constant parts*, which are typically plain texts written by developers, and *variable parts* that change with different variable values and system status. Each log consists of a certain number of tokens that are separated by delimiters, for example Adyen uses space, and the number of tokens is defined as the length of its log message.

Log abstraction methods are algorithms that remove variable parts of logs and output general patterns based on constant parts. Dimension would reduce dramatically if we use

parsed logs since the number of code lines that generate logs are much smaller than the actual number of raw logs in a system.

There are two types of methods for log abstraction, one solely focuses on parsing raw logs, and the other one needs to parse source code. Methods which only need logs as the input are discussed in this section for three reasons. First of all, log abstraction algorithms that need parsing source code have to run twice because code would change after release. Besides, such algorithms need to be modified to adjust to different languages and frameworks. The source code may not even be available sometimes. Thus the log abstraction methods that only parse raw logs are chosen for broader application usage.

### 2.2.1 SLCT: Simple Logfile Clustering Tool

Vaarandi et al. [50] introduce a clustering algorithm called *Simple Logfile Clustering Tool* (SLCT) where logs are clustered based on density in sub-spaces. First, frequent words at same positions of logs are identified as dense regions. Then a pair of cluster candidates is chosen for all combinations of frequent words. In this way, raw logs are clustered with a line pattern representative. For example, a cluster with frequent words of  $\{(1, \text{'Password'}), (2, \text{'Authentication'}), (3, \text{'for'}), (5, \text{'accepted'})\}$  would have a line pattern of "Password authentication for \* accepted". SLCT is costly for a large log data set because it is a clustering method.

### 2.2.2 A Bin-based Method

Jiang et al. [27][28] introduce a bin-based algorithm for message abstraction. The algorithm mainly consists of four steps: anonymizing, tokenizing, categorizing and reconciling steps.

Variable parts in the logs would be replaced by a generic token in the first anonymizing step using regular expressions, after which the logs are separated into different groups based on the number of words and estimated variables in the tokenizing step. Log lines with the same number of tokens and parameters would be placed in the same bin. Later, logs within each bin are abstracted into one log template, and there may exist multiple log templates in one bin. After three heuristic steps, a reconciling step is applied at the end to merge log templates that differ from each other by one token at the same position.

The algorithm outperforms SLCT in precision, recall and running time. One of the drawbacks of the algorithm, however, is that the accuracy of anonymizing should be quite high so that logs would be put into the correct bin and abstracted to the correct template.

### 2.2.3 IPLoM: Iterative Partitioning Log Mining

IPLoM [36][37] is a heuristic algorithm that uses hierarchical clustering process to parse logs into abstract events.

The first step of IPLoM is to divide logs into different groups based on the length of the tokens within logs. Logs are again put into different partitions according to tokens at the position that has the lowest number of unique values in the second step.

Later, different mappings with a set of unique tokens in two positions that have the lowest number of unique words are captured in the final step of the heuristic process. For

example, bijective relation or 1-1 relation usually indicate a strong relationship between two elements and logs with such relation would be put into one partition. As for 1-M relations or M-1 relations, the M side may be a variable or constant part, depending on the ratio between the number of unique values in the set and the number of logs that have these values at the corresponding position. M-M relations would be split into several 1-M relations.

Final message event descriptions would be created for each partition where common constant tokens are kept, and variable parts are replaced by asterisk symbols. Also, the time complexity of IPLoM is of  $O(n)$  where  $n$  is the number of inputting logs.

#### 2.2.4 HELO: Hierarchical Event Log Organizer

HELO [19] is a hierarchical process that mines a template “description” for each log partition by extracting the constant parts and variable parts from log messages. It has an offline version and an online version.

Starting with one cluster, the first step of HELO is to recursively partition logs at the token position that has the highest number of constant words in current iteration until all log groups have the cluster goodness above a certain threshold, where the cluster goodness equals to the percentage of common words in all event descriptions over the average message length. HELO considers that different tokens have different priorities: English words have the highest priority to be constant parts and numeric values, such as 166632, have the lowest priority. The authors claim that the threshold of cluster goodness should set to 40% for most log data sets. The offline version of HELO has a time complexity of  $O(n \log(n))$  and  $n$  is the number of logs.

The online version is similar to the offline version except for an extra online component. The online component checks if a new log message meets description of group templates and ends the search when it meets. If not, the cluster goodness is computed for each log clusters to see if the new message can be inserted into the corresponding cluster. If no cluster has the goodness higher than the threshold, a new cluster needs to be created. Since the goodness needs to be computed for each log cluster, the worst time complexity for  $n$  new coming logs is  $O(n^2)$ .

#### 2.2.5 Drain: Fixed Depth Tree Based Online Log Parsing Method

Drain [23] is an log abstraction algorithm that uses a fixed depth tree to parse incoming logs and clusters them into different log groups in an online streaming manner which can reduce memory usage. The basic idea is to use a tree to store the parsed logs and keep updating the tree when new logs are coming and also avoid storing all the history logs.

The first step of Drain is to use regular expressions to remove variables like pure digits and IP addresses to improve performance. Then, logs of the same length are traversed to the same first layer node from the shared root node in step two. Nodes after the second layers represent a unique token at each position of logs in the third step and logs with same token at the position are traversed toward the same node. However, the maximum number of child nodes of one node and the depth of the tree is set to a limit in advance. New coming tokens are traversed to an asterisk “\*” node instead of the actual tokens when the number

of child nodes exceeds the limit of the maximum number of children. Similarly, the leaf nodes are reached when the depth exceeds the maximum depth. Leaf nodes consist of a list of log groups. After reaching the leaf nodes, the fourth step is to calculate the similarity of log message and the log event of a log group to determine whether it should be put in the corresponding log group or a new log group. If tokens within one layer in a log group are different, the parse tree needs to be updated to asterisk “\*”.

The time complexity of Drain is  $O(n)$ , where  $n$  is the number of the logs for constant maximum depth and a constant maximum number of the child node. The authors claim Drain has better parsing accuracy and time efficiency than IPLoM. The authors make an assumption that tokens at the beginning positions of log messages are more likely to be constant parts. However, if log data fails to meet this assumption, it may hinder the performance of Drain.

### 2.3 Anomaly Detection Algorithms

There are quite some studies that focus on using log data for anomaly detection or system diagnosis. Most of the works, nevertheless, only deal with systems that are somewhat static, where no new types of logs are generated during the process. During releases, new logs are generated because of the new logging positions in source code. Related papers about log anomaly detection are listed below according to anomaly detection algorithms.

It is worthwhile to note that anomaly detection algorithms sometimes are also called outlier detection. As stated in [2], outliers are data points that can be potential anomalies and are labeled as outliers based on the different behavior from other data points, while anomalies in this thesis are more related to problematic logs. However, outliers can either be regular logs that are generated by the new code or new problematic logs that reflect some problems in the new release.

#### 2.3.1 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a dimensionality reduction method that captures patterns in high dimensional space by finding a  $k$ -dimension hyperplane, or  $k$  principal components, with a minimum squared projection error [2].

Xu et al. [53][54][55] use a PCA-based method to detect anomalies within Hadoop File System logs and Darkstar online game server logs. In [53], [54] and [55], the authors first use abstract syntax tree to parse source code and generate all possible log templates. Besides the constant string parts in logs, variable parts also carry essential information. *Identifiers* is one type of variables that identify an object or a task in the software. The *state variables* are the other type of variables that represent a certain state of an object or system, such as “COMMITTING” or “ABORTING”, “OPENING” or “CLOSING”. The identifiers are variables that are reported many times and in several log templates but have many distinct values. State variables should frequently be reported in raw logs, but the total unique number are constant even the number of logs increases. By defining thresholds for frequency and unique values, the identifiers and state variables can be found automatically in logs.

Then two types of feature vectors, *state ratio vector* and *message count vector*, are constructed from state variables or identifiers respectively. An  $m \times n$  state ratio matrix consists of state ratio vectors from  $m$  time windows, where  $n$  is the number of distinct state variables, and the value in one dimension of state ratio vectors equals to the number of appearance of a *state variable* in a time window. The constructing of message count vectors is more complicated than that of state ratio vectors. By finding *identifiers* in the raw logs, the logs with the same *identifiers* values would be grouped as one message group. For each message group, a message count vector is created, where the value at each dimension is the number of appearance of one log template. An  $m \times n$  message count matrix consists of  $m$  message count vectors, where  $m$  is the number of message groups and  $n$  is the number of log templates. In the end, PCA is used to build a  $k$ -dimensional normal subspace. The outlier scores are calculated using the distance between data points and the normal subspace.

### 2.3.2 Support Vector Machine (SVM)

Support Vector Machine (SVM) is a classifier that minimizes the margin. In anomaly detection or other binary classification problems, margin is defined as the smallest distance between the decision boundary and data points from two classes [41].

Kimura et al. [31] extract four features from each log template of network logs: frequency, periodicity, burstiness and the ratio between the number of observed logs during monitoring and the number of logs in a whole period for each log event obtained in log abstraction. The authors then use a supervised SVM with the Gaussian kernel to find anomalies.

### 2.3.3 Decision Trees

As stated in [21], a decision tree is a classifier that has a flowchart-like tree structure, where an input goes through the nodes and branches and eventually is classified at the bottom of the tree in a leaf node. Inside a tree structure, each internal node represents a test on an attribute, and each branch represents a result of the corresponding test. In the end, each leaf node holds a class label for the input [21]. When used in anomaly detection, the leaf node in a decision tree indicate whether it is an anomaly or not.

Reidemeister et al. [43] improve the SLCT clustering algorithm from [50] by mining frequent regions, which are tokens in log messages, and applying further clustering based on Levenshtein distance between the frequent regions. One common subsequence of tokens is then chosen as a representative pattern for each log cluster. Later, the authors treat each log file within an hour as a finite sequence and use a bit-vector to encode the log file. Each dimension in a vector represents one log template from SLCT, and the value is 1 if one log template exists in the current log file. At last, a decision tree is applied to find recurrent faults in the system using log files generated by fault-injection experiments.

Bose et al. [6] use decision trees and class association rules to select proper features from individual log events and sequence features in X-ray logs. The authors treat each log event associated with one activity or action as one data point and construct feature

vectors. They also suggest using clustering,  $k$ -nearest neighbor and one-class support vector machines to assist with class labeling using instances with known labels.

### 2.3.4 Finite State Automation (FSA)

Finite state automaton (FSA) or finite state machine (FSM) is an abstract model with a limited number of possible states, inputs, outputs, transitions from a state to an other and actions performed to each transition [32]. It can be used to model log event sequences and find anomalies.

Mariani et al. [38] build FSA models to detect anomalous event sequences in logs. They first generate log events using SLCT and rewrite concrete variable parts of events to the order ID based on the appearance order of concrete variables. Then event sequences are used to build FSA. Event sequences which differ from the ones observed in previous executions would be considered as anomalies. Fu et al. [18] use similar method to build FSAs to find the low performance issues in log sequences by counting the execution time of each state transition in state sequences.

### 2.3.5 Hidden Markov Model (HMM)

According to [29], Hidden Markov models (HMMs) is an extension of the FSA model. When using an HMM to assign label to each unit in a sequence, the HMM computes a probability distribution over possible label sequences and find the most possible label sequence [29].

Yamanishi et al. [56] leverage HMM mixtures which is a linear combination of HMMs to detect anomaly in network logs in real-time. Each component in HMM mixutres represents a specific log pattern. They train HMM mixtures with different number of components at the same time and choose the best one. For each incoming sequence of logs, an outlier score is computed based on the probability and any log sequence with a score above a threshold would be labeled as anomalous. Salfner et al. [45] applied *Hidden Semi-Markov models* (HSMM) to detect anomalies in telecommunication system after clustering the log sequences.

### 2.3.6 Associative Rule Mining

Association rule mining is a type of data mining technique that can extract frequent patterns, associations or causal structures within a data set [58]. Lou et al. [35] build linear relationships among logs to detect anomalies. The authors first group log events based on the parameter values and build program invariants from log message counts in each log message group. Invariants are conditions in the software that always hold in normal running situations. For example, the number of logs that indicate open a file should be equal to that of close a file. The authors use either a brute force searching algorithm or a greedy searching algorithm for invariants mining. When the expected computation complexity is high, the authors choose the greedy algorithm over the brute force algorithm. Any new log that does not meet the requirement of related invariants is considered as anomalous.

Huang et al. [25] use associative classification methods to find anomalies in log sequences. In feature extraction, they compare the abstracted log model with the non-abstracted model. They choose SLCT as the log abstraction method. As for the non-abstracted model, the “word-columnID pairs” are used to build the feature matrix. For instance, the log “Open the file in resources” is transformed into “Open-1 the-2 file-3 in-4 resources-5”. The feature matrix is of  $M \times N$  dimension, where the M is the number of word-columnID pairs that have occurred more than once, and N is the number of logs in the log file. Their experiment result shows that applying the abstraction method SLCT would improve the performance of classifiers compared with the non-abstracted model.

### 2.3.7 Multi-Instance Learning (MIL)

In [47], the authors treat the log anomaly detection problem as a *Multi-Instance Learning* problem where logs of one week are considered as a bag and logs of each day are considered as multiple instances in the bag. Also the bag is only considered as normal if all instances inside the bag are not anomalous. More than 300 to 400 features are extracted from logs using bootstrapped feature selection. This work may be too general for logs that do not contain enough features. Moreover, this method takes much manual work to select the features for the classifier.

### 2.3.8 Clustering

Clustering is a popular unsupervised method in the field of anomaly detection [2]. It would assign similar objects into the same cluster and detect anomaly based on the clustering result. Kc and Gu [30] are able to detect anomalous instance in a cloud system. The authors first use hierarchical clustering based on *Message Appearance Vector*, which is a binary vector, in each instance from the distributed system. The small clusters are defined as anomalies. Furthermore, anomalies in large clusters are detected by using the nearest neighbor technique based on graph edit distance in *Message Flow Graph* in each instance.

Shang et al. [46] process the Hadoop logs into log sequence sets without repetition and permutation and show the new sequence sets generated in production environment compared to experiment environment. However, the number of resulting sequences that is shown to the developers may still be a lot. To solve this, Lin et al. [34] improve this method by considering the history sequence. After getting the log sequence sets, the authors assign weight to each event. The less frequent events which are measured by Inverse Document Frequency are of more importance. Moreover, the events showing up only in the production environment are of more importance. Then the different sequence sets are used to construct weight vectors, which is then further used by hierarchical clustering. Outliers are labeled when the distance between a new production cluster with all old clusters exceeds a certain threshold.

### 2.3.9 LSTM-based Methods

A *long short-term memory* (LSTM) network is a *Recurrent Neural Network* (RNN) that is able to capture long-term dependency on inputs [57].

## 2. RELATED WORK

---

Zhang et al. [57] extract *tf-idf* features for each pattern of logs in a time window and constructed a feature vector based on this. They treat the anomaly detection as a binary classification problem, and their LSTM network outperforms other state-of-art supervised learning classifiers such as Logistic Regression, SVM and Random Forest in the log data set collected from a web server cluster and a mailer server cluster.

Taking the timestamp and parameter values which exist in the logs into consideration, Du et al. [15] have trained an LSTM neural network to detect not only execution path anomaly but also parameter value and performance anomaly in log sequence generated from Hadoop. Also, the algorithm also needs to parse the source code.

### 2.3.10 Statistical Methods

Besides machine learning methods, some papers use a more intuitive statistical method to find anomalies in logs. According to Lim et al. [33], one can detect anomalies simply getting into log frequency by inspecting the slope, average and maximum in the time windows for simple systems such as telephony systems. While for a more complex system, this method may not work because there may be more correlated logs.

Nagaraj et al. [39][40] present a method for performance analysis in a distributed system, they divide logs into two types: *event* log messages and *state* log message. The authors use a t-test to find different distribution between two log sets and dependency networks to model the relationship between *events* and *states*. The final anomaly result would be shown to users in a direct graph plot indicating the cause. The main disadvantage of this method is that heavy manual works are needed to divide log messages into *events* and *states* in advance. Another more sophisticated method can be found in [20], the authors can find execution flow outliers and execution performance outliers after constructing log events for each log statements from source code and tracking log sequences.



## Chapter 3

---

# Feature Extraction

Logs mainly consist of plain-text attributes that are not suitable to be processed directly by machines. Extracting meaningful features is essential for downstream anomaly detection algorithms and the cornerstone of this thesis project. This chapter mainly discusses the construction of feature vectors from logs within the release process using Adyen log data set. First, we go over log structure in Adyen and explore log behaviors during release. Moreover, we use log event sets before and after release as data set, and choose to detect anomalous log sequences within release. In the end, based on the chosen features, assumptions and definitions are proposed in the last section.

### 3.1 Data: Logs in Adyen

As mentioned before, the primary goal of this thesis is to design a pipeline that can detect anomalous logs, which may indicate a problem in a host during release. To this end, it is necessary to explore the log data first. Adyen mainly uses *Log4j*, a Java logging framework, to generate log statements that are stored in hosts. The logs are later stored within *LogSearch* in a JSON format. We sometime refer a raw log as a **log entry**.

#### Log Structure

Each log consists of multiple fields, and several important fields are listed below. Although Adyen uses *Log4j* for logging which provides a somewhat structured log framework for *Java*, the most significant parts of the logs, *message* and *extra* attributes, are still in plain text. Some fields are optional like *extra* and *PSP reference*. Besides, each log does not contain the information about the logging line in the source code, which means raw logs need to be processed first to find the exact logging positions. An example of a log in Adyen is shown as in Figure 3.1, though some information has been left out deliberately.

- Host: host name of the log.
- App: application name of the log.
- Class name: Java class name of the log.

### 3. FEATURE EXTRACTION

---

- **Severity:** severity that the log has. Either of severity INFO, WARN or ERROR. Most logs are of INFO level, less than 0.3% of logs generated are of WARN and ERROR level. Usually Adyen developers assign WARN for logging problems outside company, such as wrong or missing values in certain forms from merchants, and assign ERROR to problems that caused by platform itself.
- **Timestamp:** timestamp when the log is generated.
- **Message:** main part of the log and is usually of plain-text written by developers. It consists of variable parts and constant parts.
- **Extra (Optional):** stores the extra information when message is too long, usually is of JSON format or traceback from the compiler.
- **PSP reference (Optional):** currently 16 digits that represent a specific task along the platform, can be considered as a task ID that is shared by a few log entries.
- **Thread ID:** sometimes is the same as the PSP reference, sometimes is the generally Thread name that generates the log, which can be shared by quite a lot of different logs. For example, more than twenty thousand logs are shared with one common thread ID in one host from a log data of 30 minutes.



























Field	Action	Value
@timestamp	 	2018-05-13T20:35:52.378Z
app	 	<input type="text"/>
app-group	 	<input type="text"/>
classname	 	<input type="text"/>
dc	 	<input type="text"/>
extra	 	<div>Exception: No <input type="text"/> Data</div> <div>#011at #011at #011at #011at #011at #011at #011at</div>
host	 	<input type="text"/>
message	 	<div>Exception: No <input type="text"/> Data</div>
pspreference	 	<input type="text"/>
severity	 	ERROR
tags	 	<input type="text"/>
threadid	 	<input type="text"/>
type	 	<input type="text"/>

Figure 3.1: A log detail from *LogSearch* in Adyen. The fields and their value of log entry are shown in this figure. Some values are hidden because of security reasons.

## 3.2 Parsing Logs or Not?

It is vital to extract appropriate features from logs as the input of anomaly detection algorithms. There are different anomaly detection algorithms, and they require different ways of constructing feature vectors, such as log sequences or time windows. We usually use the term **message** to indicate the combination of message and extra attributes of Adyen logs in the scope of the thesis. In fact, we use message and extra together as the input of Iterative Partitioning Log Mining algorithm (IPLoM, Section 2.2.3) as shown in Figure 4.2.

Depending on whether to parse the logs with log abstraction methods, there are a few possible options. One of the options is to consider all logs in a host during the release as a long discrete sequence and find anomaly sub-sequence within it. If we consider the order of the logs in the sequence, we need to deal with logs with nearly the same timestamp as online systems usually have multiple threats generating logs in parallel. Alternatively, we can use log counting vectors which record the number of different log events within each time window. The log counting vectors have  $M$  dimensions, where  $M$  is the total number of log events and each dimension in the log counting vector represents the occurrence number of one type of log event. A time window is considered anomalous when the counting deviates significantly from history.

The following sections describe some intuitive thoughts in extracting the most appropriate features from release logs. There are mainly two ways of extracting the features from the logs. One is to parse the raw logs and the other to use other fields of the logs along with some other system information. Since this project focuses on using logs only, all extracted features are confined with the information obtained from logs.

### 3.2.1 Not Parsing Logs

If we choose not to parse the logs, we have to use combinations of other attributes in raw logs such as app, class name or severity to distinguish different log events.

#### Log Counting Vectors

There are generally multiple threads from different applications running at the same host in Adyen, which would generate logs in a concurrent mode. As shown in figure 3.2, there are high correlations among the log frequencies from different *Java* Classes. The data are from all the hosts during one release. Each host has different applications running on them and logs on each host are always from the *Java* Classes in each host. It is more reasonable to build a model for each host considering the different distribution of logs.

It is possible to do an anomaly detection only based on the attributes like *app*, *class name* and *severity* from the logs. The combination of these three fields can be considered as one feature, for example  $(app_1, classname_1, INFO)$ ,  $(app_1, classname_2, INFO)$  and  $(app_2, classname_3, WARN)$  are all different features and the number of occurrence of logs with the same combination in a time window is recorded as the corresponding feature value. One  $m$  dimension vector is built based on the  $m$  different combination of *app*, *class name* and *severity* for each time window per host.

### 3. FEATURE EXTRACTION

---

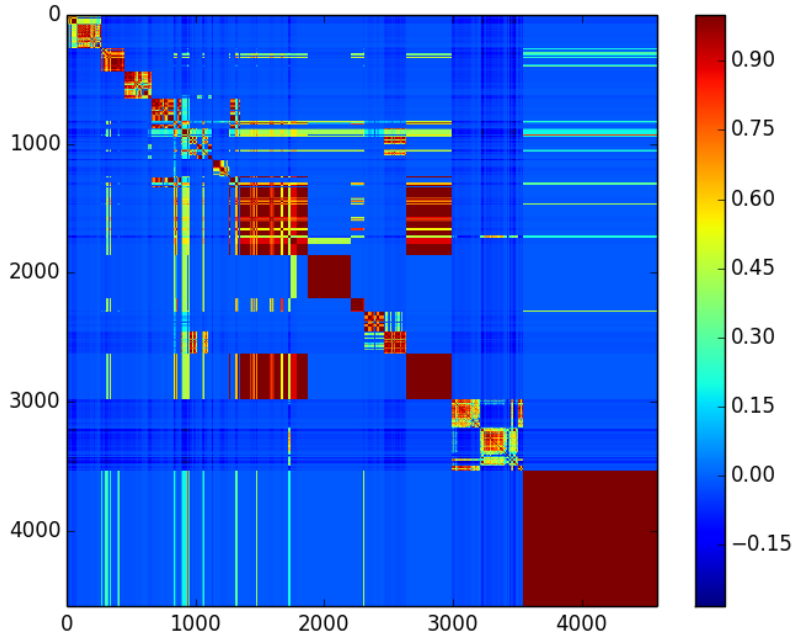


Figure 3.2: A heat map of the correlation matrix of occurrence number of logs from different classes in Adyen. If the color of one point in the heat map gets close to red, it indicates that the two classes are positively correlated, meaning logs from the two classes tend to show up together. When the color of one pixel gets close to dark blue, this means the two classes are negatively correlated, meaning the number of logs from one class tends to decrease when logs from the other class occur.

Time windows before the release belong to the history set, and we can use the history set to build the anomaly detection model. Time windows after the release are considered as anomalies when they do not fit into the model. Some experiments based on the toy example, nevertheless, have shown that such model would result in a very high false positive because there are always logs from new app and class name generated when the code updates. The intuitive idea is to assign different weights to features. However, expertise is again needed to determine which features are more important than the rest. Merely assigning a higher weight to features with a higher level of severity would also result in poor result according to the toy data set.

The benefit of not parsing logs is that we can extract features directly from other fields of logs. However, there is also cost of not parsing log messages: it may not guarantee sufficient granularity to distinguish different log events for anomaly detection because same class may have several logging points.

#### 3.2.2 Parsing Logs

The granularity of the anomaly analysis is better when parsing the log messages than using log counting vectors described in 3.2.1 because one *class name* may be able to generate multiple different log templates. In [25], the authors state that log abstraction methods would improve the performance of anomaly detection classifiers and reduce the size of data to analyze. Thus, we decide to use log message abstraction in preprocessing step. Also each template represents a type of *event*. In the scope of the thesis, log event and log template are used interchangeably.

Same template may be from different *app* and *class name*, this may either be the developer using the same log statement in two different places or the different Java Class execute the same code to generate the code. While there is no way to distinguish the two reasons without parsing the source code. In the experiment, the method abstraction algorithm takes only the concatenation of the fields *message* and *extras* as input for parsing to get fewer templates.

Each log entity, as shown in Figure 3.1, is now associated with one log template that represents a kind of system behaviour or status. With the other attributes like *timestamp*, *host*, *PSP reference* and *thread ID*, we can extract features based on parsing results. More detailed information of parsing logs is discussed in Chapter 4.

#### 3.2.3 Conclusion

In the thesis, log templates obtained by parsing raw log messages are chosen to build the features. The *message* and *extra* attributes provide most information among all attributes, and utilizing the two features would result in better granularity. It makes no sense to neglect the most informative part of the data set.

### 3.3 After Parsing: Extracting Features from Release Logs

After preprocessing the logs into *events*, the dimension would generally decrease dramatically. For example, in Adyen data set five million of logs could be parsed into around one thousand events in one experiment using IPLoM. After this, features need to be built for the release procedure and use the feature vectors to capture anomaly behavior.

There are different feature models for anomaly detection, i.e., sequence anomaly detection, time series anomaly detection, or normal feature vector anomaly detection. To construct a proper feature model, some factors need to be taken into consideration.

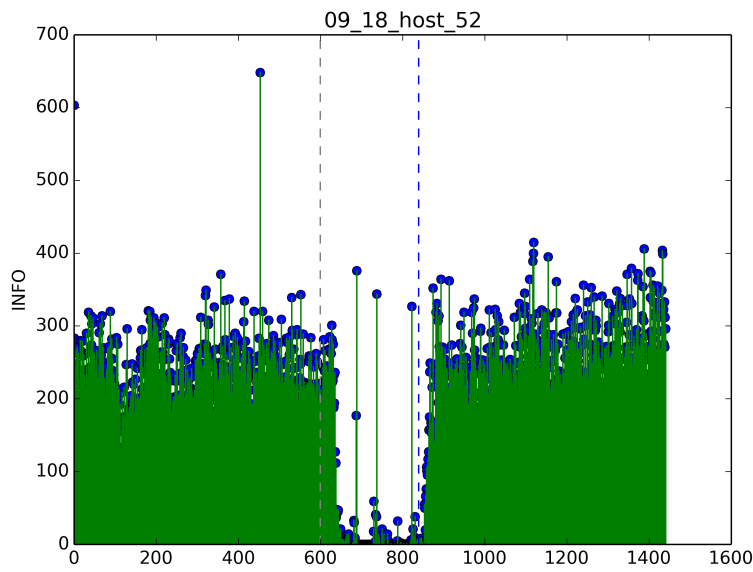
#### 3.3.1 Release Sections and Log Data Sets

During a release the host behaves differently from the ordinary working status. Logs related to the release are generated while other tasks are halting or pausing. As shown in figure 3.3, the release process can be divided into three rough time sections.

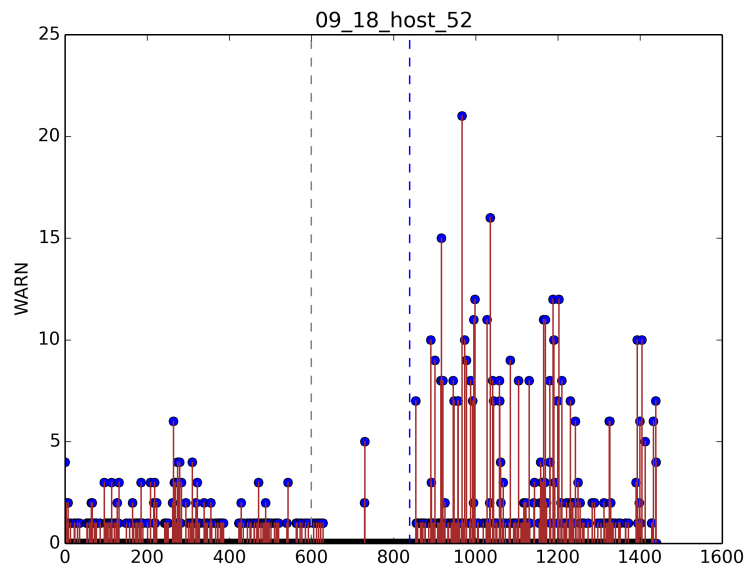
- **Before release:** the host would download the release code and be taken out of load-balancer.

### 3. FEATURE EXTRACTION

---



(a) The frequency of INFO logs per second.



(b) The frequency of WARN logs per second.

Figure 3.3: The frequency of INFO and WARN logs per second within one host in a release. The x-axis is the time in seconds and y-axis represents the number of logs of one severity per second. The grey dashed line represents the time when the host is taken out of load-balancer, and the blue dashed line represents the time when the host is brought back to load-balancer

- **During release:** the host would install the release and start software.
- **After release:** the host would be brought back to load-balancer.

There may exist anomalies in two sections: the *during release* and the *after release*. In the *during release section* fewer logs are generated because on-going tasks are halted or stopped and new tasks stop coming. After finishing the release, the host is back to load-balancer, restarting the tasks that are paused before and enabling new features. Thus more logs would be generated in the *after release section*.

To detect anomalies, we need to split each log based on its time stamp and put them into different data sets. We refer to the release that needs to be monitored as **current release**. For better reducing the impact of noise in release, **previous releases** can also be collected in advance and be included in the history log data.

Here, we define two types of log data:

- **History log data:** Logs that are generated before the grey dash line in Figure 3.3, including the *before release section* in current release and all logs in previous releases.
- **New log data:** The *during release* and *after release* sections in current release.

As mentioned above, now the task can be rephrased as: we use the history log data to detect anomalous logs in new log data.

#### 3.3.2 Noise in Release Logs

The logs provide rich information about status and performance of hosts. Nevertheless, there is noise during the release that would mask the actual anomalies and make a successful release look like an abnormal one. The authors of [2] state that noise lies within the “boundary between normal data and anomalies”. As in the case of monitoring release logs with anomaly detection, noise is a collection of logs different from history log data, but actually is novel patterns related to release process. For instance, the *WARN* logs related to connection time-out complaints that would not hinder the performance, an *ERROR* logs that happened before or the new *INFO* logs related to release may all be considered as noise points. One of the intuitive solutions is to get rid of duplicate log events and only look into new log events.

According to the experience of developers, most logs are not anomalies. Moreover, directly using the event frequency as features would yield an unsatisfactory result when we apply popular anomaly detection approaches such as *One-class SVM* or *PCA*, according to our explorations. The possible reason is that the frequency of different events is somewhat random in the release. Therefore taking the noise into consideration would hinder the performance considerably. It is also stated that the message appearance vector is more robust than message frequency feature because some normal log message happens very frequent and has nothing to do with anomalies [30].

#### 3.3.3 Utilize Temporal Attributes: Time Windows

Each Log is generated in a specific timestamp as shown in Figure 3.1. It is natural to utilize this temporal attribute of logs in anomaly detection. We can build features from logs within certain size of time windows during release. There are few features that can be used to construct feature vectors. For example, the number of occurrence of log events during release, or important statistic information fields, such as connection time or payment processing time, in some log events. However, given the fact that normal points behave very differently in three release sections, models need to be robust to the sudden change of behavior or three separate models need to be built for these three sections.

Besides, each time window is now labeled as normal or anomaly, which requires the labeling to be very accurate. For instance, we may need to take into consideration that if the release is anomalous, whether we should consider all time windows during release and after release as anomalies or only consider time windows which consist of an anomalous event as anomalies. The duration of the time windows also needs to be determined. Moreover, time windows during release may have much fewer logs generated than the other two release sections. Thus, the length of time windows within *during release section* might need to be larger.

In fact, finding the anomaly log sequences is sufficient for debugging and monitoring purpose during release. Determine which time window is behaving normally or not is more suitable for daily system monitoring when the hosts are more stable as in some papers [53][54][55]. Moreover, showing developers the anomalous log time window can be a problem because there may exist hundreds of log events in one time window causing it difficult for developers to spot the errors immediately.

In the following sections, we discuss methods that can be used for anomaly detection with time windows.

#### Prediction-based Methods

During exploration some logs are observed to be periodic, such as generating reports in the back office the internal system in Adyen for further data analysis. Some other logs related to payment tasks are more of a stochastic process, which would be generated on the request of the merchants and would go through a series of procedure, giving the logs a long-term dependency across multiple hosts in the platform. However, the latter type of logs which belong to one particular payment method or merchant may also have a specific pattern. For example, the frequency of the logs within a particular day in a week may have a value close to a constant and any deviation from the mean value is an indicator of an anomaly.

One could use time series to model logs with certain patterns and to detect abrupt changes in log behaviors, such as counting the occurrence of certain log events in a duration of time, by using regression methods such as ARMA [8]. Other logs do not have this kind of behavior, but can be occasionally seen and not affect the system behavior much, which should be viewed as noise that needs to get rid of in order not to affect the accuracy of the model. Nevertheless, different time series model needs to be built for every type of logs. Expertise is needed to choose which type of logs are more decisive during release.



#### Distance-based Methods or Linear Methods

Besides regression methods, we can also leverage nearest neighbor distance method or PCA for anomaly detection of time windows [2]. The nearest neighbor distance method is similar as we described in the Section 4.6.2, and PCA-based methods were already discussed in Section 2.3.1. There are also other possible algorithms for temporal anomaly detection, we refer interested readers to Chapter 9 in [2].

#### 3.3.4 Anomaly Detection in Log Sequences

As mentioned in Section 3.3.3, instead of looking for anomalies in time windows, we choose detecting anomalous log sequences directly.

In its logging framework, Adyen uses three customized log severity levels: *INFO*, *WARN*, and *ERROR*. As mentioned in Log Structure of Section 3.1, *INFO* logs are usually normal logging statements. *WARN* logs are usually indicator of outside problems, and *ERROR* logs are logging problems caused by payment platform itself. In release monitoring, nevertheless, we cannot determine whether logs are related to release problems directly based on severities. Not all *WARN* logs or *ERROR* logs are the indicators of release problems. Besides, *INFO* logs may also represent bugs in a release. In manual log monitoring procedure, developers tend to dive into *ERROR* and *WARN* logs and try to find out if they are generated by, or related to new functions in the platform. Sometimes they would also search for log messages with keywords such “exception”. If these logs are found caused by new code in release, it means they are probably release related.

*Logness*, the internal log clustering tool in Adyen, would send a notification to the developer when a new *WARN* or *ERROR* log event occurs. The key lies on how to distinguish which log event is more critical or problematic quantitatively if several new *ERROR* or *WARN* events occur in a release. We can mine context information in log sequences such as frequency of events and timestamps of the first occurrence of the events to assign different weights to log events. The log sequences consist of log entries with the same task ID, for example, same *PSP reference* or *thread ID*.

Each log sequence can be represented in an  $m$ -dimensional vector, where  $m$  is the number of total log events in the data set. The vector is either frequency-based or appearance-based vector. Frequency feature vector records the number of occurrence of each event in the log sequences, and appearance feature vector consists of values of either 1 or 0 depending on the appearance of log events within sequences.

During release monitoring, developers in Adyen send notifications in public channel once they spot one problematic log. The anomalous sequences are sequences that consist of such anomalous log events. There are two types of anomalous log sequences:

1. Log sequences consist of at least one new event.
2. Log sequences consist of only old events.

Usually, new log events that have never been seen before release are generated by new log lines due to updating of code. Anomalous log sequences consist of new events are easily

being spotted. The new log sequences with only old events sometimes may also indicate problems because release in other hosts might cause the anomalous event occurs before the current host starts releasing.

After forming log sequences by tracking task IDs in logs, we can construct binary vectors and later assign weights to each event. The details of calculating weights and simplifying log sequences are introduced in Chapter 4.

#### 3.3.5 Supervised or Unsupervised

While it is normal to solve anomaly detection in an unsupervised way, it is also possible to use a supervised algorithm. It is hard to treat the problem directly as a binary classification problem because we have a very biased data set without enough anomalies. Although there are other methods to deal with the bias distribution of different categories like cost-sensitive learning and adaptive re-sampling [2], we choose to use unsupervised methods for anomaly detection of log sequences.

This project has focused on using unsupervised methods for anomaly detection. Proximity-based algorithms are used to compare with the work in [34]. Using a supervised algorithm can, however, lead to future projects that could build a more accurate pipeline and update the labels each time when new data points are labeled.

### 3.4 Definitions and Assumptions in Anomaly Detection

As stated before, the goal of this thesis is to develop a pipeline that uses anomaly detection algorithms to capture the most problematic log event sets during release. This section presents the fundamental definitions and assumptions that we made in anomaly detection. Two types of log data sets used in anomaly detection are first introduced. Later, assumptions of the anomalies are made based on requirements of release monitoring. We also propose the definitions of anomalies and outliers in the scope of the thesis.

#### 3.4.1 Data Sets

As already discussed in Section 3.3.1, based on timestamp logs from one host can be divided into two data sets: *history log data* and *new log data*.

All logs are parsed and transformed into a log sequence by putting logs with the same PSP reference or same thread ID into one sequence. Log event sequences are further simplify into *log event sets* by removing permutations and duplicates, reducing the number of the log sequences in analysis. The message appearance vector is used because of its robustness to noise [30]. All log event sets in new log data are compared with the sets in the history log data to compute a quantities outlier score. More details of the process can be found in Chapter 4.

### 3.4.2 Assumptions

The underlying assumption is that the logs in history log data should be considered as normal or contain very rare anomalies because a host is usually of normal state before starting its release according to our observations. This is the essential requirement for anomaly detection being used in release monitoring. If the majority of logs in history log data are not normal data, the unsupervised methods proposed in this project does not work, and a binary classification algorithm can be used if the logs are all labeled. Moreover, new logs are compared with the history to determine whether there is an anomaly.

One of the assumptions of anomalies is that the previously unseen log events provide more information than those old log events [34]. Another assumption is that log events with low frequency also have more discriminative power than those with high frequency [34]. This is based on the experience of developers that most regular log events should be generated by routine procedures. Rarity indicates that the logs belong to a unique process and needs more attention.

Moreover, log events with higher verbosity levels like WARN or ERROR also provide more information than those with lower verbosity level. We propose this assumption of log severity to address the fact that different severity levels have different discriminative power. There may also be cases when INFO log events would be labeled as anomalies by developers. In most cases, however, the anomalies are still of WARN and ERROR severity levels according to our observations.

We believe that log events that provide more information or have more discriminative power would also be more likely to be anomalies. Some of the assumptions may not be true during a release. We use two artificial data sets to test the robustness of the pipeline when one of the assumptions does not hold in Chapter 5. For example, when an anomalous event is of INFO or an anomalous event is first seen before release, we would like to see if the pipeline is still able to detect it.

### 3.4.3 Definition of Anomalies and Outliers

An anomaly is defined as a log event set that has a significant deviation from the previous *history log data* and is indicating a problem during release. Those with a significant deviation from the previous *history log data* but not problematic are considered as noise. During release two types of noise might occur: the first one is new normal log templates added after the release, and the other one is recurrent error log messages that are not related to release. Noise and anomalies together are called outliers.



## Chapter 4

---

# Pipeline of Monitoring Release Logs

This chapter mainly introduces the design of the pipeline for monitoring release logs, which is based on the recent work of [34] and [46]. An illustrative diagram of the pipeline is shown in Figure 4.1. In the first step of the pipeline, history log data is used to build an anomaly detection model after logs being preprocessed into log event sets. Later the anomaly detection model is used to filter new log data. **Anomalous event sets** are log event sets with an outlier score greater than the filter threshold. **Important known sets** are log event sets with an outlier score less than the filter threshold but have an average weight (Equation 4.19) greater than the weight threshold. Filtering important known sets are a complementary step of anomaly detection that helps capture anomalies that occur before release. In the final steps, potential anomalous event sets and known important event sets would be clustered separately and displayed according to a ranking function.

Though the original work in [34] and [46] are designed for monitoring errors in production environment based on the logs in the test environment of big data applications such as Hadoop, their frameworks are also suitable for release monitoring because it can capture the log anomalies in a different status based on the history log data.

The main contributions of this work are improving the pipeline in [34] by adding extra severity weight of log events, and preprocessing steps of log sequences, making the pipeline suitable for monitoring Adyen release logs, a more complex release log data. Moreover, we test the pipeline with three anomaly detection algorithms on Adyen logs.

There are some essential problems that we need to solve when extracting features from Adyen logs. For example, log data sets that are used in [34] and [46] only have one severity, which is simpler than Adyen log data set that has three verbosity levels, including INFO, WARN, and ERROR. Moreover, compared with the rest of the log sequences, some logs in Adyen can form surprisingly long sequences due to general thread IDs. Those differences require extra preprocessing steps before anomaly detection and solutions to deal with such problems are also included in this chapter.

In [34], the authors first cluster new log data and then calculate outlier scores of each cluster by measuring its distance with the nearest cluster in history log data. This method is not suitable in Adyen data set because calculating outlier score based on clusters instead of each log event set would provide poor granularity that masks true anomalies. Besides anomaly detection, one additional step is added to capture **important known event sets**

#### 4. PIPELINE OF MONITORING RELEASE LOGS

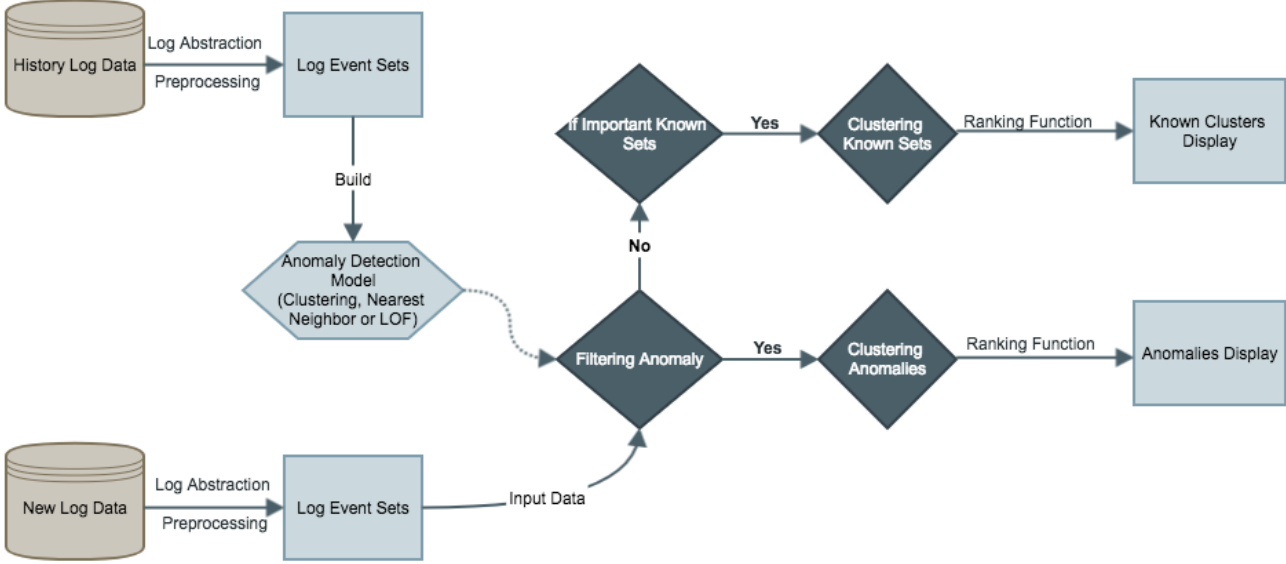


Figure 4.1: The diagram of pipeline of monitoring release logs. Logs from one host are divided into two data sets based on timestamp. New log data consists of logs from *during release section*, and *after release section* within current monitoring release. The history log data consists of previous releases and *before release section* of the monitoring release. The logs within two data sets go through the same preprocessing step and are abstracted into log events. Later log event sequences which consist of log entries with same task ID are further simplified to log event sets. Unsupervised anomaly detection algorithms are used to detect anomalous event sets in new log data according to the deviation from history log data. Any log event set from new log data with an outlier score higher than a threshold would be considered as a potential anomaly, and clustered to show to developers in the end according to a ranking function. Log event sets with an outlier score lower than threshold but important according to the average weight of the events would also be clustered, and shown to the developers according to the same ranking function.

that happen again in the monitoring release that may be neglected by the anomaly detection algorithms. Important known clusters mainly consist of event sets in new log data that are similar to that of history log set but have a high average weight, indicating the event sets might include important events, such as WARN events, ERROR events, or rare INFO events. In the end, a ranking function is chosen to print out the clusters that are most likely to reflect problems in the release.

A more detailed explanation of each step is listed as follows in each section. Since different hosts have different applications and different distribution of logs, each host should have its anomaly detection model. All the following parts are done host-wise.

## 4.1 Collecting Log Data

Logs of two **previous releases** and current **monitoring release** in one host are collected in advance. Related attributes of each log entry, including *timestamp*, *PSP reference*, *Thread ID*, *host*, *message* and *extra*, are stored in the experiment for further usage in the process. A unique ID is assigned for each log entry for further reference task. In this experiment, we choose to collect logs of the previous two releases. It is also possible to collect more releases if possible, we stick with two releases mainly because of efficiency because too many logs would slow down the log abstraction method and anomaly detection process.

There are two data sets in the pipeline. All the previous releases logs and logs before releasing in the monitoring release altogether are considered as *history log data*, and all the logs after release are called *new log data*. The basic idea is to use collected *history log data* to predict anomalous event sets in *new log data*.

In real application, old logs should be cleaned up after new logs are collected because of efficiency reasons.

## 4.2 Parsing Logs to Templates

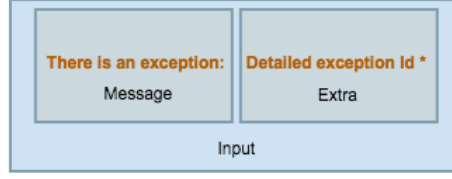
In this section, we discuss the first log message abstraction step in the pipeline: preprocessing of raw logs, abstraction, and its reconciling step.

**Templates** can also be called **events** as they represent an event of logging system status or certain actions. There are generally two ways of parsing logs to templates: using source code along with raw logs, or using solely raw logs. Parsing source code to generate log events would be more efficient and accurate than abstraction methods that use solely raw log messages because the abstraction methods that utilize source code can avoid ambiguity by getting code information to distinguish variable parts and constant parts in raw logs. However, experiment results in Chapter 5 show using raw logs from log abstraction methods already have high accuracy and satisfying performance of anomaly detection in Adyen release logs. In this work, we focus on one message abstracting method that solely relies on input of raw logs. Other methods can be explored in future work.

When talking about **log messages**, we are referring to two attributes in Adyen log structure: *message* and *extra*. Because *extra* in Adyen is initially part of *message*, the *message* attribute is concatenated with *extra* attribute together with a space delimiter as the input for the log abstracting algorithm as shown in Figure 4.2. In this figure, *Message* attribute, “There is an exception:”, along with *extra* attribute, “Detailed exception Id \*” where an asterisk replaces pure digits ID, are concatenated into the final input “**There is an exception: Detailed exception Id \***” for log message abstraction.

*Extra* attributes are optional and do not exist in all logs. Besides, if *extra* attributes have multiple lines, only the first two lines are put into the concatenated string. All *message* attributes with a token number exceed a length limit, which is 300 for the Adyen data set, would be truncated to have better efficiency. The number 300 is a reasonable choice because most log templates in Adyen have far less token numbers than that.

After concatenation, logs would be processed in advance to remove evident variable parts in the inputs. Then, IPLoM (Iterative Partitioning Log Mining) is applied to logs from



$$\text{Input} = \text{Message} + \text{Extra (Optional)}$$

Figure 4.2: An input example of concatenation of message and extra attributes for the message abstracting method IPLoM (Iterative Partitioning Log Mining). The message and extra are both plain-text attributes in a log entry. The extra is an optional attribute and is generated when the original message is too long and get truncated into message and extra. In this example, “There is an exception:” is the message, and “Detailed exception Id \*” is the extra with the pure digits replaced by “\*”. The final input for this log entry is “There is an exception: Detailed exception Id \*”, where the message and extra are concatenated and separated by a space.

both history log data and new log data to obtain log templates. Finally, similar logs are clustered together and considered as the same template to further correct the possible error in the message abstraction process.

All logs from both *history log data* and *new log data* now belong to one specific log template, and the number of features is significantly reduced, for example in one release the three million logs are abstracted into two thousand log templates.

### 4.2.1 Prepossessing the Logs

At this step, variable parts such as pure digits or special symbols in raw logs are replaced with a wildcard “\*” symbol by simple regular expressions beforehand to improve the performance [22]. Also, JSON parts can also be parsed in advance to get the meaningful parts only because they have very different structures from plain-text strings and not suitable for message abstraction. We use the keys of JSON parts to construct a plain-text string, and use it as an input of message abstraction. Different preprocessing methods need to be applied to achieve a better abstraction result for other log data.

Pure digit parts in logs from Figure 4.3, for instance, can be replaced in advance. The five raw log messages would have the same anonymizing result “Example function (\*) should not be bigger than \*!” before abstraction, which would prevent message abstraction methods putting them into different templates. Another preprocessing example in Figure 4.4 shows how keys from JSON format are extracted and listed in the final result. The preprocessing of JSON is based on the idea that keys are more meaningful and can be considered as constant parts.

We try to avoid complicated regular expressions in this step as regular expression package in Python can be extremely slow because it uses backtracking instead of using finite automata-based techniques [13]. Moreover, the number of raw logs is significantly greater



than the number of parsed log templates. Using regular expressions directly on raw logs is more efficiently than using regular expressions on the reconciling step in Section 4.2.3.

---

```
1. Example function (35) should not be bigger than 31!
2. Example function (40) should not be bigger than 31!
3. Example function (32) should not be bigger than 31!
4. Example function (33) should not be bigger than 31!
5. Example function (41) should not be bigger than 31!
After anonymizing:
1. Example function (*) should not be bigger than *!
2. Example function (*) should not be bigger than *!
3. Example function (*) should not be bigger than *!
4. Example function (*) should not be bigger than *!
5. Example function (*) should not be bigger than *!
```

---

Figure 4.3: An example of anonymizing pure digits with wildcards in log messages. Five log messages and their anonymizing results are shown in this figure. The digits before log messages are log IDs used for reference. As we can see, after anonymizing, five log messages have the same format: “Example function (\*) should not be bigger than \*!”. They belong to the same template even before log abstraction.

---

```
JSON: {"currency": "EURO", "link": "https://www.example.com", "method": "PAY",
"response-time": 210, "status": "SUCCESS"}
After extracting keys:
JSON: currency link method response-time status
```

---

Figure 4.4: An example of extracting keys from JSON format logs. A log message and its preprocessing result are shown in this figure. We extract the keys from the JSON part, and list them based on the original order within in JSON.

#### 4.2.2 Log Message Abstraction Method: IPLoM

There are many algorithms[18][19][28][36][37][50] for abstracting the logs to obtain log events. *Iterative Partitioning Log Mining* (IPLoM)[36][37] is chosen in this step to abstract logs in the pipeline. IPLoM is based on heuristic rules and scales linearly with the number of logs [36][22]. IPLoM first divides logs with the equal number of tokens in its message into one cluster. Then different clusters are partitioned according to the position with the least number of unique words, and logs which have the same unique word in that position would be put into one partition. At last, partitions are further split based on different mappings found in the same position. The output results of IPLoM are a list of abstracted log templates, as well as corresponding log IDs in each template. Log templates represent the

#### 4. PIPELINE OF MONITORING RELEASE LOGS

---

pattern of logs within the partition. As shown in Figure 4.5, if tokens at certain position in the partition are the same, the templates would use the common token at the corresponding position, such as “Payment”, “does”, “not”, “contain”, and “ID”. However, if there are multiple unique tokens at one position, the templates would use “\*” to represent for various unique tokens. In this example, “\*” is used to represent for “merchant”, and “shopper”.

---

```
1. Payment X does not contain merchant ID
2. Payment Y does not contain merchant ID
3. Payment X does not contain shopper ID
After abstraction:
Payment * does not contain * ID
```

---

Figure 4.5: An example of log message abstraction. Three log messages and their template are shown in this figure. The digits before log messages are log IDs that used for reference. In the template, positions where all logs have same common word would keep the common word in the corresponding positions. For example, “Payment”, “does”, “not”, “contain”, and “ID” are all kept in the template. At the 2nd token position, there are two unique tokens: “X” and “Y”. Similarly, at the 6th token positions, there are multiple unique words among logs: “merchant” and “shopper”. Thus the template has a “\*” symbol at the 2nd and 6th token.

Compared with the data set used in original IPLoM work [36][37] or the evaluation paper [22], Adyen has a more complicated log structure. We have logs from different severities, apps, or class names. Logs of different severity levels are abstracted separately to avoid them being put into one log event. Logs of INFO severity are abstracted by IPLoM, generating INFO templates. While WARN logs are also abstracted by IPLoM in parallel at the same time and we obtain WARN templates as well.

Logs with different app and class names, however, are not abstracted separately. We observe that logs of different apps or class names can have same log template. They may either be from two same logging line in source code, or Java methods calling the same logging line from different applications. We still treat these logs with different *apps*, or *class names* as same event considering log messages are the essential feature in logs.

The accuracy of abstraction in this step would affect the performance of the anomaly detection. When IPLoM assigns logs generated from the same log line into different templates, we can merge similar templates in the reconciling step. Nevertheless, if logs from different events are assigned with the same template, we need to add another separating step to split templates. Great care must be taken to avoid putting two different logs into one template. The suggested parameters of IPLoM in [22] are used, and the result turns out to be very strict for the Adyen data set. In the experiment of 14 releases in Table 5.1, usually few different logs are put into the same template, but many logs from the same log code line are in several templates because of the strict parameters. Further analysis of the message abstracting result is shown in Chapter 5.

Sometimes similar templates are not merged into one template in IPLoM, which may

hinder the performance of the anomaly detection algorithms. The first leading cause is that some similar templates are of the same length. The other cause is that IPLoM does not recognize some variables. It is not possible to use more complicated regular expressions on the raw logs directly because of the vast number of raw logs. Because the number of log templates is far less than the number of raw logs, it is more reasonable to apply the complex regular expressions to the log templates instead. Thus a reconciling step is added after IPLoM to improve efficiency and to avoid duplicate templates in the final output.

### 4.2.3 Reconciling Clustering after IPLoM

In this section, we propose a reconciling step to merge templates that are very similar. We first use regular expressions to further process log templates and then use a hierarchical clustering on templates based on the defined similarity in Equation 4.2.

#### Regular Expression

First more complex regular expressions are leveraged to find variable parts in templates and to replace them with wildcards “\*”. These variables include file paths, web URLs, email addresses, dates, timestamps, and other complicated parameters. This step is similar to the example shown in 4.3, but instead of using regular expressions for matching variables in raw logs, we apply them to log templates.

#### Similarity of Templates

Moreover, the similarity between two log templates is calculated according to Equation 4.2.  $X, Y$  are log templates.  $|Wild|$  is the number of positions that are represented by wildcard asterisks “\*”.  $|X|$  and  $|Y|$  are the number of tokens in the templates.  $X_i$  is the token from  $X$  at the index of  $i$ .  $equ(X_i, Y_i)$  equals to 1 when token  $X_i$  and  $Y_i$  are the same, and none of them is asterisk. The defined similarity is to calculate the same token at each index.

$$equ(X_i, Y_i) = \begin{cases} 1, & \text{if } X_i = Y_i \text{ and } X_i \neq * \text{ and } Y_i \neq * \\ 0, & \text{otherwise} \end{cases} \quad (4.1)$$

$$Similarity(X, Y) = \frac{\sum_{i=1}^{\min(|X|, |Y|)} equ(X_i, Y_i)}{\frac{|X| + |Y|}{2} - |Wild|} \quad (4.2)$$

#### Clustering of Similar Templates

*Logness* clusters logs together if the length of the longest common subsequence is greater than 0.6 of the whole length of the message [17]. We also use 0.6 as the similarity threshold of clustering. Because wrongly clustering logs together would affect the performance of anomaly detection, any similarity lower than 0.6 would be assigned to 0. Later, agglomerative hierarchical clustering is used with the same similarity threshold to cluster templates obtained from IPLoM together. Moreover, the template with the smallest ID in one cluster is chosen as the representative template for the cluster.

An example of reconciling clustering is shown in Figure 4.6, where each cluster is separated by a dashed line. The digits at the beginning of the cluster are the cluster IDs, and the digits in front of log templates are the template IDs. In Cluster 5, for example, there are four templates: Template 62, 78, 91, and 315, among which Template 62 is the representative template. All logs that used to belong to Template 78, 91, or 315, would now be considered as Template 62 in further steps of the pipeline.

Noted that the similarity calculated by Equation 4.2 is stricter than the definition of [17] to prevent any wrong clustering. In fact, no templates are falsely clustered together in our training set. Details of performance of this step can be found in Chapter 5. The time complexity of calculating the similarity matrix is  $O(m^2)$ , where  $m$  is the number of templates. Because some templates are already the same after removing remaining variables, the similarity can be reused before actually being calculated.

### 4.3 From Log Sequence to Binary Feature Vector

In this section, we describe the step of creating binary feature vectors from the log sequences for *history log data* and *new log data*.

First, log sequences are formed with logs with same task ID. Compared with the rest, some log sequences may be much longer because general thread IDs are continuously generating logs. Long sequences would be more likely to be considered as outliers because they have much more log templates than normal sequences. The long sequences are cut into short sequences using a specific length threshold, making all sequences comparable in length.

As already mentioned before, authors in [30] state that message appearance feature is more robust than message frequency feature because some normal log messages happen very frequently and have nothing to do with anomalies. Thus we ignore the permutations and duplicates in the log event sequences and construct log event sets based on log event sequences. There may exist the same log event set within *history log data*, or *new log data* respectively. We only store unique log event sets in data sets. Occurrences of log event sets are considered, while the occurrence frequencies of event sets are ignored. Each log template is one feature, and every log event set can be represented by an  $m$ -dimension binary feature vector where  $m$  is the total number of log templates. If there is one template in the log event set, the value of the corresponding dimension in a binary vector is one.

Eventually, the *history log data* or *new log data* would have different log event sets in the form of binary feature vectors, and can be used to create weighted feature vectors.

#### 4.3.1 Task ID

Task ID can either be *PSP reference* attribute or *thread ID* attribute from log entries. Logs with the same PSP reference, which is a 16 digit length at the moment that can be viewed as a task ID in Adyen, belong to the same log sequence. Almost 88% of the logs has a PSP reference in our data set. For the rest without a PSP reference, a thread ID is used as the task ID. Thread IDs can be very specific and still considered as the task IDs. However, general thread IDs such as “main”, “thread\_1”, or “Processor\_2” are not confined to

### 4.3. From Log Sequence to Binary Feature Vector

---

---

```
Cluster 1
320: Modified new id= * (was null) for Shopper * reason: *
321: Modified new id= * (was *) for Shopper * reason: *
-----

Cluster 2
318: Process with type * and ref= * scheduled with id= *
319: Process with type 'example' and ref= * scheduled with id= *
-----

Cluster 3
161: Processing * Label: example id: * as Successful with reason: * * * * *
317: Processing X* Label: example id: * as Successful with reason: [example]
-----

Cluster 4
152: Processing * Label: * id: * as Successful with reason: * *
316: Processing * Label: * id: -* as Successful with reason: *
-----

Cluster 5
62: ListDetails for example1= * example2= *
78: ListDetails for example1= * example2= * *
91: ListDetails for example1= * example2= * * *
315: ListDetails for example1= * example2= * * - ***
-----

Cluster 6
321: DateDetail * * (% of total time)
313: DateDetail Reader[ * (% of total time)
-----

Cluster 7:
308: Search: regex * matches * allow: false
309: Search: regex .*X.* matches EXAMPLE allow: false
-----

Cluster 8:
306: PathName: * ; using fold: *
307: PathName: * ; using example: *
-----

Cluster 9:
304: Stopping task with label *
305: Stopping task with label 'Example'
```

---

Figure 4.6: An example of template clustering. Clusters are separated with dashed lines. Each cluster is shown with its cluster members, and the digits before log templates are the assigned template IDs.

one specific task. Therefore, they can generate surprisingly long sequences compared with specific thread IDs or PSP references.

### 4.3.2 Cutting Long Log Sequence

Log sequences with general thread IDs mentioned in Section 4.3.1, may consist of up to 30,000 log entries and last for one hour in the experiments. Long sequences need to be split into subsequences to make them comparable to other ordinary log sequences.

There is no guarantee that log entries in a log sequence are stored in a sorted time order because logs are extracted from several different log files in the host. It is thus not able to cut the long sequences directly because of the unordered sequence. For cases where ordered log sequences are possible, the following sorting step is optional and can be cut directly based on the chosen length threshold.

Sorting the long list of length  $n$  is too time-consuming because it needs a time complexity of  $O(n\log(n))$ . However, a complete sorting of the long log sequences can be avoided by dividing all log entries into different groups based on its rounded timestamp. Logs generated in the same minute are put in one sub-list. This splitting is reasonable because almost all log sequences with a regular length only last for less than 10 seconds. The sub-lists can be sorted with an overall time complexity of  $O(n\log(l))$ , where  $l$  is the average expected length of the sub-list and usually is smaller than 100 in the experiments.

After sorting the long sequences, logs can be safely cut into regular length sequences using a threshold that equals to the 99th percentile of the length of log sequences.

### 4.3.3 Log Event Set: Further Simplifying Log Sequence

There are still many log event sequences after the previous section. To further decrease the amount of input data for anomaly detection, log event sequences need to be simplified. Thus we remove all duplicates and permutations of the event sequence, keeping unique combinations of log events. In other words, all log sequences are converted into unordered log event sets. Although this simplification step ignores the extra information containing the order and occurrence numbers of log templates within the sequence, the appearance feature of the log event is sufficient for anomaly detection as shown in Chapter 5.

### 4.3.4 Binary Feature Vector

If we obtain  $m$  templates after IPLoM and its reconciling step, we use  $m$ -dimensional binary vectors to represent unordered log event sets. The *history log data* and *new log data* would each have a set of binary feature vectors and can be used further to create weighted feature vectors. All binary vectors are rather sparse because on average each log sequence consists of less than 10 templates, and usually there are more than one thousand templates during release in each host.

## 4.4 Weighted Feature Vector

Ignoring the fact that different log events have different importance from the perspective of release may lead non-ideal performance of anomaly detection. For instance, when compared with a rare log template indicating unexpected interruption of the current payment, a frequent generated log event that marks the successful ending of the payment is less significant in anomaly detection in release logs. Thus log templates should be assigned with different weights.

Three types of weights are calculated for every template, and their final weights are calculated using Equation 4.7. Three types of weights align with our assumptions of importance of log events in anomaly detection mentioned within Section 3.4.2. The first two weights are calculated the same as in [34]. The weights are listed below:

- The first weight is the frequency weight using inverse document frequency (*idf*) to measure how often one template occurs in different sequences. Templates which appear in fewer log sequences have more discriminative power and should have higher weights [34].
- Another weight is the contrast-based weight to show whether one template is first seen after release. Templates which first occur after release would have more discriminative power and thus have a higher contrast weight [34].
- The last one is the severity weight. It is based on the assumption that the higher the severity, the more likely a log template reflects a potential problem or failure of the release. Therefore, templates with higher severity should have higher severity weight.

### 4.4.1 *idf*-based Event Weight

Inverse document frequency (*idf*) was originally proposed in [48] and is widely used in information retrieval field to show how important a word is considering the number of documents where a word shows up. Quite some papers [12][42][44] argue that *idf* is not only a heuristic feature but also has a good theoretical justifications from statistics, information theory and other perspectives.

The events that appear less in log sequences would be regarded as rarer and have a greater chance to reflect an anomaly during release [34]. For event  $i$ , the *idf*-based event weight are calculated as below,

$$w_{idf}(i) = \log \frac{N}{n_i + 1} \quad (4.3)$$

where  $N$  is sum of total number of task IDs, and  $n_i$  is number of task IDs where the event  $i$  appears. In the experiment, the base of 2 is used. In fact, the logarithm base is not important according to [44]. The *tf-idf* (term frequencyinverse document frequency) is not used here because we want to prevent assigning a higher weight to frequent events. Moreover, *idf* is more align with the purpose of release monitoring, which is to find rarely occurred suspicious log events. There may be cases when the anomalous log event during

release occur frequently. However, compared with normal log events, their frequencies are still not high. We find anomalous event confined to limited task IDs during a release, generally fewer than 10, or even only 1. Therefore, we use *idf* instead to focus on more discriminative events with a relatively low frequency.

### 4.4.2 Contrast-based Event Weight

New events that have not appeared before the release can either be noise, i.e., a new function in the system, or an anomaly that represents problems during release. Either way, new log events are more discriminative in finding possible anomalies in the monitoring release [34]. Contrast weight aligns with the experience of developers that new log events after release are more relevant to release and provide more information than old events.

$$w_{contrast}(i) = \begin{cases} 1, & \text{if } i \notin \text{History} \\ 0, & \text{otherwise} \end{cases} \quad (4.4)$$

where *History* represents the *history log data* that consists of all three time sections mentioned in 3.3.1 of previous releases, as well as *before release section* in the current monitoring release.

### 4.4.3 Severity Event Weight

Logs with different severities have different importance. According to experience of developers, generally the higher the severity, the more likely it would be the problematic logs during the release. Thus different weights are assigned to logs with different severities as shown in Equation 4.5, where the severity ratio is a constant number that should greater or equal to 1. Moreover, a grid search is conducted in Chapter 5 to find the suitable severity ratio based on Adyen data set.

$$w_{severity}(i) = \begin{cases} 1, & \text{if } i \text{ is INFO} \\ 1 \cdot \text{severity ratio}, & \text{if } i \text{ is WARN} \\ 1 \cdot \text{severity ratio}^2, & \text{if } i \text{ is ERROR} \end{cases} \quad (4.5)$$

### 4.4.4 Overall Weight

In the original work of [34], the overall weight of each event is calculated as follows:

$$w(i) = 0.5 \cdot \text{Norm}(w_{idf}(i)) + 0.5 \cdot w_{contrast}(i) \quad (4.6)$$

We add severity event weight to this equation. A weighted feature vector is used to represent each log event set. When considering the severity, the overall weight of one event *i* is calculated by

$$w(i) = w_{severity}(i) \cdot (0.5 \cdot \text{Norm}(w_{idf}(i)) + 0.5 \cdot w_{contrast}(i)) \quad (4.7)$$

where *Norm* is a Sigmoid function for normalizing the *idf*-based event weight:

$$\text{Norm}(x) = \frac{1}{1 + e^{-x}} \quad (4.8)$$



After abstraction, each sequence is transformed into an  $m$ -dimensional binary event appearance vector, where  $m$  is the number of all events after applying message abstracting methods. We note the binary vector as  $v_{binary}$ . Its values are set to 1 only if corresponding events appear in the log sequence, otherwise 0. The final weighted event appearance vector  $v_{weighted}$  is calculated as follows

$$v_{weighted} = v_{binary} \cdot w_{weight}^T \quad (4.9)$$

where  $w_{weight}$  is also an  $m$ -dimensional vector, and value in each dimension is the overall weight of each event.

## 4.5 Distance and Similarity

A distance metric needs to be chosen to quantify the difference in log sequences for the proximity-based algorithms, such as clustering and nearest neighbor. Given the fact that the matrix consisting of weighted vectors is rather sparse and that the log event sets should be shown to the developers in a ranking of according the possibilities of indicating a problem, this pipeline is very similar to information retrieval: log event sets are documents, and log events are words. Finding anomalous event sets during release is similar to finding relevantly new documents using the old documents data set. In this light, we test distance metrics that are popular in document retrieval in Chapter 5 to see which one would perform the best in release monitoring. Related work about metrics is listed below.

In the original pipeline [34], Lin et al. use cosine distance in the paper without comparing to other distance metrics.

Aggarwal et al. [3] state that Manhattan distance performs better than Euclidean distance in high dimensional space k-means clustering and that fractional distance is even better at preserving the meaningfulness of proximity measure. Huang et al. [24] compare Euclidean distance, KL divergence, Pearson coefficient, cosine similarity and the Jaccard coefficient on text document clustering and finds that Jaccard and Pearson coefficient measures find more coherent clusters while average KL divergence, Pearson coefficient would give more balanced clusters. Choi et al. [11] conduct a survey about 76 binary similarities and divided them into 3 categories: Correlation Based, Non-Correlation Based and Distance-Based.

In Chapter 5, we test six distance metrics, including Cosine, Pearson, Jaccard, Euclidean, Hamming, and Manhattan.

## 4.6 Anomaly Detection

After building weighted feature vectors, two more steps that include filtering and clustering are used to find anomalies in log event sets. We also propose complementary steps of anomaly detection when anomalies occur before release. The two steps include filtering **important known sets**, and clustering them. The important known sets are log event sets that are similar to history log data but they are still important and provide information about release. The details of **important known sets** are described in Section 4.7.

Providing feedback of anomaly detection is essential. Instead of labeling log event sets as normal or anomalous directly, we assign each event set with an quantitative outlier scores to indicate their chance of representing release errors. Though various types of anomaly detection algorithms exist, proximity-based algorithms are tested for the sake of simplicity and used to compare with the benchmark [34]. In anomaly detection, data points are *occurrence feature vectors* of log event sets. We describe details of filtering and clustering in three anomaly detection algorithms respectively: hierarchical clustering (Section 4.6.1), nearest neighbor distance (Section 4.6.2), and LOF (Section 4.6.3)

### Filtering Anomalies

The first step is filtering anomalies. An anomaly detection algorithm is used to calculate outlier scores that measure how far log event sets (data points) within *new log data* deviate those from *history log data*. Log event sets with an outlier score which exceeds a **filter threshold** are captured as potential anomalies. The filter threshold is calculated as a percentile of the outlier score as shown in Equation 4.6.1, Equation 4.6.2, or Equation 4.17. The filter threshold is a local outlier factor value for LOF algorithm, but is a distance for clustering and nearest neighbor distance.

For those log event sets with an outlier score lower than the filter threshold, they are usually consider similar to data in history log data and called **known event sets**. We also filter them according to a threshold on average template weight (Equation 4.19), and obtain **important known event sets**.

### Clustering Anomalies

After obtaining potential anomalies, we use a hierarchical clustering algorithm to cluster anomalous event sets, where log event sets with distance lower than **anomaly cluster distance** are clustered together. A representative log event set in each cluster is chosen in a way that the representative event set should have the minimum average distance to the rest event sets in a cluster and can be considered as the center of the cluster [34].

Similar to clustering anomalies, we also cluster important known event sets using the same anomaly cluster distance. **Known clusters** and **potential anomalous clusters** would be sorted and shown to developers in the end, according to a ranking function described in Section 4.7.1.

#### 4.6.1 Agglomerative Hierarchical Clustering

As already mentioned in section 4.5, each log event set can be regarded as a document, and the result can be viewed as relevant document clusters in an information retrieval system. The purpose of clustering here would be very similar to document clustering in this sense.

There are two popular clustering algorithms for document clustering: one is  $k$ -means clustering and its variants and the other one is hierarchical clustering. Although hierarchical clustering is of quadratic time complexity, the value of  $k$  need to be determined before running the  $k$ -means clustering, and it may vary from the different hosts and different releases. Besides, hierarchical clustering can provide a more hierarchical structure for analysis while

result from  $k$ -means is flat. Thus agglomerative hierarchical clustering is chosen in our experiment.

There are some commonly used linkage algorithms for hierarchical clustering that are used to determine how distance is calculated between sets of observation, such as single linkage, complete linkage, average linkage, and Ward's linkage. [52]

According to [52], definitions of linkages are listed as follows: The distance between two clusters calculated by single linkage is the distance of two closest objects from the two clusters. On the contrary, the complete linkage uses the farthest distance of the objects within the two clusters. The average linkage uses the average distance between all pairs of objects within the two clusters. The unweighted version of average linkage is called unweighted pair group method average (*UPGMA*). There is also a modified version, a weighted pair group method average (*WPGMA*), which considers the number of objects in two clusters and uses them to calculate the weighted average distance. The Ward's linkage is to minimize the increase of the squared errors within one cluster.

Complete linkage is chosen in the pipeline because complete linkage is more robust to outliers and not sensitive to the shape and size of clusters according to [4]. Also, authors in [16] state that complete linkage has better retrieval effectiveness in document retrieval application.

### Clustering History Log Data

First, log event sets from *history log data* are clustered using a **first cluster distance** and a center is chosen as the representative center of the cluster [34]. The representative event sets not only are used to calculate the outlier score in the filtering procedure but also represent its potential anomalous cluster in Section 4.7.

### Outlier Score

The distance between a log event set and a cluster is defined as the distance between the log event set and the representative event set within the cluster [34]. For each data point in new log data, there is one closest representative event set and the corresponding minimum distance is recorded as an **outlier score** and is calculated by [34]

$$Outlier(\bar{Y}) = \min_{\bar{R} \in \mathbb{R}} \{dist(\bar{Y}, \bar{R})\} \quad (4.10)$$

where  $\bar{Y}$  is one data point in *new log data*,  $\bar{R}$  is a representative log event set for one cluster,  $\mathbb{R}$  represents a collection consisting of all representative log event sets in history log data,  $Outlier(\bar{Y})$  is the outlier score of data point  $\bar{Y}$ .

We assume that data points within *history log data* are normal log events, and  $c$  number of clusters are obtained after cluster. For  $n$  log event sets in *new log data*, a distance matrix with the shape of  $n \cdot c$  dimension can be computed and the minimum distances are derived from the matrix, resulting an outlier score matrix of  $n \cdot 1$  dimension, which is noted as **outlier score array (OA)**.

Zeros in the outlier score array show that the corresponding data point is exactly at the center of the clusters. After exclude those zeros, **filter threshold** and **cluster distance** can

be computed from outlier score array using the **filter percentile** and **cluster percentile** as percentile separately:

$$\mathbf{Filter\ threshold} = \text{Percentile}(\text{non-zero } OA, \text{ filter percentile}) \quad (4.11)$$

$$\mathbf{Cluster\ distance} = \text{Percentile}(\text{non-zero } OA, \text{ cluster percentile}) \quad (4.12)$$

However, using a constant value for the filter threshold and cluster distance is also possible, which would be discussed more in Chapter 5.

### 4.6.2 Nearest Neighbor Distance

Nearest neighbor is a supervised algorithm that determines the label of one data point based on the labels of its nearest neighbors. We use **nearest neighbor distance** to denote its unsupervised version. Nearest neighbor distance algorithm is similar to clustering algorithm, instead of clustering *history log data* in the beginning, each data point in *history log data* is used directly to calculate the distances. The average distance to  $k$  nearest neighbors of each data point in *new log data* is recorded as outlier score as shown in Equation 4.13, which is more robust than using single  $k$ th distance according to [2].

$$\text{Outlier}(\bar{Y}) = \frac{\sum_{i=1}^k \text{dist}(\bar{Y}, \bar{N}_i)}{k} \quad (4.13)$$

where  $\bar{Y}$  is one data point in *new log data*, and  $\bar{N}_i$  is one of  $\bar{Y}$ 's  $k$  nearest neighbors.

The non-zero outlier score array ( $OA$ ) is constructed similarly to clustering in Section 4.6.1. Afterward, a filter threshold is calculated by Equation 4.11. Data points with an outlier score exceeding the threshold would continue with the next step for clustering anomalies.

Ideally, nearest neighbor distance algorithm would have a better granularity than clustering algorithm because instead of measuring one nearby event set cluster, nearest neighbor distance considers  $k$  closed neighbor event sets, though clustering has a better time efficiency.

### 4.6.3 LOF: Local Outlier Factor

Nearest neighbor distance and clustering all tend to detect anomalies that have a higher distance from the normal data points. However, when the distribution of the data varies dramatically in different parts of feature space, distance-based algorithms might fail to capture the anomalies [2]. Local Outlier Factor [7] is a density-based algorithm that can detect anomalies based on the local density of data points.

The original LOF algorithm detects anomalies in one data set by computing local density of one point compared with other points in the data set. Here we make a slight modification of denotations in [2] that allows us to use history log data to find anomalies in new log data. A history data point in history log data is denoted by  $\bar{X}$ , and a new data point in new log data is denoted by  $\bar{Y}$ .  $D^k(\bar{X})$  is the  $k$ -nearest neighbor distance of the history data

point  $\bar{X}$ , which only considers neighbors in the history log data.  $L_k(\bar{Y})$  denotes a collection of history data points within  $D^k(\bar{Y})$ . The reachability distance, the essential concept of LOF, of  $\bar{Y}$  respect to  $\bar{X}$  is defined as the greater value of  $D^k(\bar{X})$  and the distance of  $\bar{X}$  and  $\bar{Y}$  [2]:

$$R_k(\bar{Y}, \bar{X}) = \max\{dist(\bar{Y}, \bar{X}), D^k(\bar{X})\} \quad (4.14)$$

The reachability distance in Equation 4.14 can be asymmetric for  $\bar{X}$  and  $\bar{Y}$  because the  $D^k(\bar{X})$  may not equal to  $D^k(\bar{Y})$  [2]. In the next step, we calculate the average reachability distance of  $\bar{Y}$  by computing the mean value of the reachability distances of  $\bar{Y}$  and history data points  $\bar{X}$  that are within its  $k$ -nearest neighbor distance  $D^k(\bar{Y})$  [2]:

$$AR_k(\bar{Y}) = \text{MEAN}_{\bar{X} \in L_k(\bar{Y})} R_k(\bar{Y}, \bar{X}) \quad (4.15)$$

The reachability density of  $\bar{Y}$  is defined as  $1/AR_k(\bar{Y})$  [7]. Finally, we use the average value of average reachability distance of  $\bar{Y}$  divided by the average reachability distance of history data points  $\bar{X}$  that belong to  $L_k(\bar{Y})$  to compute local outlier factor of  $\bar{Y}$  [2]:

$$LOF_k(\bar{Y}) = \text{MEAN}_{\bar{X} \in L_k(\bar{Y})} \frac{AR_k(\bar{Y})}{AR_k(\bar{X})} \quad (4.16)$$

### Outlier Score

We use LOF directly as outlier score:

$$Outlier(\bar{Y}) = LOF_k(\bar{Y}) \quad (4.17)$$

The higher the LOF value, the higher the outlier score and the chance a data point would become an anomaly. The filter threshold can be computed similarly to Equation 4.11. The cluster distance, however, needs to be assigned with a constant value.

## 4.7 Final Output of Potential Anomalous Clusters and Known Clusters

After the filtering step and clustering step, we obtain a collection of potential anomalous clusters. We would also obtain known clusters similarly as the complementary output. However, given the large log data in Adyen, the number of clusters after the two steps is still large, sometime more than a thousand. A ranking function is used to show the most problematic clusters among the collection for manual inspection.

We first propose two ranking functions in Section 4.7.1. Then we use ranking functions to sort potential anomalous clusters in 4.7.2. Finally we leverage known clusters to capture potential anomalies that first occur before release in Section 4.7.3.

### 4.7.1 Ranking Functions

A ranking function needs to be used to sort the collection of final potential anomalous clusters and should reflect the possibility that a cluster may indicate a problem for the release.

For log data set in Adyen, we propose two ranking functions. The first ranking function is outlier scores computed from anomaly detection algorithms as shown in Equation 4.18, representing how dissimilar the new log event set to the log event sets in history log data. The other intuitive option is the average weight of events in each log event set, which is shown in Equation 4.19. The average weight is the mean value of the weight of events within a log event set and shows the importance of the log event set. For each potential anomalous cluster or known cluster, the ranking score is calculated by its representative log event set.

$$\text{Ranking function 1 : } R_{set} = \text{Outlier}(set) \quad (4.18)$$

$$\text{Ranking function 2 : } R_{set} = \frac{\sum_{i \in set} w_i}{|set|} \quad (4.19)$$

where the  $set$  is the representative log event,  $|set|$  is the number of events in the log event set,  $i$  is an event within the log event set and  $w(i)$  is computed by Equation 4.7.

As shown later in Chapter 5, the average weight function in Equation 4.19 would achieve a better result than outlier score in capturing of anomalous clusters. We only use average weight (Equation 4.19) in the ranking of known clusters because the original design purpose of known clusters is to capture log event sets that are not that different from event sets in history log data but still important.

### 4.7.2 Potential Anomalous Clusters

A maximum of 10 potential anomalous clusters with the highest ranking score are shown with a detailed representative log event set to the developer for manual inspection whether it is an anomaly. The number, **10**, is chosen to decrease the manual work as much as possible. This choice is further supported by experiment result in Chapter 5 that 10 is enough to capture most anomalous event sets. Most anomalies should be captured already in the potential anomalous clusters.

### 4.7.3 Known Clusters

All algorithms mentioned above mainly deal with log event sets that vary greatly from history log data. However, there may exist potential anomalies that happen right before the release and can be overseen by anomaly detection algorithms. A complementary process is designed to capture the known event sets that may indicate recurrent old errors in the system. It is an intuitive process that uses the average weight (Equation 4.19) to show the top 10 known clusters in monitoring release. If developers cannot find anomalies in potential anomalous clusters, they can check top 10 of known clusters. Again the number **10** is sufficient enough to capture known clusters according to Chapter 5.

## Chapter 5

---

# Experiment Results

This chapter mainly describes experiments that help us to choose the suitable parameters and evaluate the performance of our monitoring pipeline, enabling us to address and answer the research questions proposed in Chapter 1.

First we introduce our experiment setup in Section 5.1, including data sets, labels, benchmark, and evaluation metrics. The log data we collect directly from Adyen is the **original data set**, which is used to test the performance of the pipeline in general cases. However, anomalies in the release sometimes do not meet our assumptions. For example, anomalous logs are of INFO severity instead of higher severity levels, or logs related to release problems occur before the release rather than after release in one host. Two types of **artificial data sets** are built based on the original data set to test the robustness of the pipeline and answer **RQ3** (Research Question 3). Thanks to developers in Adyen, we can label anomalous log events during release and use them to evaluate the performance of anomaly detection algorithms.

To answer **RQ2**, parameters and distance metrics are then evaluated using the clustering algorithm in this chapter. Later, the parameters of the nearest neighbor distance algorithm and LOF are also determined by experiments. In the end, the test set is used to evaluate the performance of anomaly detection algorithms to answer **RQ1**.

All experiments are conducted host-wise. In other words, we use our pipeline to monitor release logs from the monitoring host and display potential anomalous clusters sorted by the possibility of indicating a releases problem in that host. Although logs with the same PSP reference can appear across multiple hosts within the platform in Ayden, experiment results show that log data from the monitoring host is adequate for monitoring release.

## 5.1 Experiment Setup

### 5.1.1 Data Sets

Release log data from hosts is collected in advance for experiments. The **training set** includes 14 problematic log events from 13 separate releases of live hosts as shown in Table 5.1. The **test set** as shown in Table 5.2 consists of 10 problematic log events from 7 releases. Moreover, log data from previous two releases of the same host is also collected to better

## 5. EXPERIMENT RESULTS

---

remove noise points during the release. We use the training set to choose parameters in the pipeline and test set to evaluate the actual performance of the anomaly detection algorithms. Release log data of each host is processed separately in the pipeline. In other words, the experiments are done host-wise. We use only logs which belong to the monitoring host as the input of the pipeline.

During our observations most anomalous log events are generated within 20 minutes after the release. Although there are a few examples when an anomalous log event is first spotted an hour later than the release, developers can always rerun the pipeline when trying to find suspicious log event sets outside the 20-minute scope. Therefore in our experiments we always choose a *before release section* and an *after release section*. These two sections last both 20 minutes. The length of *during release section* varies from a few minutes to half an hour, depending on when it is taken out and brought back to the load-balancer.

Within each release section mentioned in 3.3.1, we choose a starting time of 20 minutes before the specific host is taken out of the load-balancer and an ending time of 20 minutes after the host is back to load-balancer. The logs generated between the starting time and ending time of the release would be collected. We divide the collected logs into two types of data set: new log data and history log data. **New log data** includes the logs generated in the *during release section* and *after release section* of the monitoring release. **History log data** includes logs that are generated within the *before release section* from the current monitoring releases and all log data in three sections from previous releases.

As shown in Table 5.1 and Table 5.2, we show information of the release instances that are used in the training set and the test set. A **release instance** in every row includes the monitoring release and its two history releases in one host. Each release instance is assigned with a release instance ID. There might be multiple anomalous log events in a monitoring release, such as Release Instance 1, 14, 15 and 16. We list severity of all anomalous log events of the release instance in the second column. The total number of logs of different severity levels are shown from the third to the fifth column. We use # to denote the number of logs of one severity. For instance, # INFO represents the number of INFO logs in one release instance. We notice that INFO logs usually account for the majority of raw logs in release instance.

In Table 5.1 and Table 5.2, most anomalies are of severity WARN or ERROR. To test the performance of anomaly detection algorithms for INFO anomalies, artificial *INFO anomalies data set* is generated and used in experiments. The experiments are done with the original data set shown in the Table 5.1 and Table 5.2, unless mentioned explicitly using artificial data sets.

### Artificial Data Sets

We propose assumptions in Section 3.4.2 that log events have different importance in helping us to find problems in the release. Log events that are of high severity or unseen before release are believed to have a higher chance of indicating problems during the release. It is quite essential to consider cases when the assumptions do not hold. Two artificial data sets are constructed based on the original data set to test the robustness of anomaly detection algorithms if one of the assumptions is not true.



Release Instance ID	Anomalous Log Event Severity	# INFO	# WARN	# ERROR
1	WARN	269273	1167	110
2	WARN	3561232	22714	81
3	WARN	535771	2157	235
4	WARN	50168	98	0
5	INFO	22822	292	4
6	INFO	25132	324	0
7	WARN	4492877	83317	1
8	ERROR	1082705	739	216
9	WARN	4815530	36353	1
10	WARN	1774316	1523	13
11	WARN	513840	108	0
12	WARN	527054	93	0
13	WARN	78325	253	0

Table 5.1: Release instance information in training set, including 13 monitoring releases and 26 history releases from 12 hosts. Each row in the table represents one release instance, consisting of a monitoring release and two history releases in one host. Release instance ID is a manually assigned ID for each monitoring release and its two history releases. Each monitoring release might consist of a few anomalous log events, such as Release Instance 1. The number of logs within the release instance of each severity are listed separately using the # symbol. For example, # INFO represents the number of INFO logs in the three releases of the corresponding release instance.

- **INFO Anomalies Data Set**

Since most anomalies in the data set are of WARN or ERROR severity, the *INFO anomalies data set* is constructed by making all anomalous log events into INFO severity. If the anomalous log event is already of INFO severity, such as Release Instance 5 and 6, this release instance would not be included in this data set. Moreover, this data set also does not consist of release where the anomalous log events are accompanied with other WARN or ERROR log templates in an event set because the remaining higher severity logs would still make it has a high average weigh.

- **Recurrent Anomalies Data Set**

It is possible that an anomalous log event is both in history log data and new log data. This makes it hard to capture anomalies by filtering anomaly step for its low outlier score. The starting time of the new log data is also at the start of the during release section when the host is taken out of the load-balancer. The *recurrent anomalies data set* is constructed by postponing the starting time of the new log data right after the first occurrence time of corresponding anomalous log event so that it would have a contrast weight of 0 in Equation 4.4 instead of 1. If there are multiple anomalous log events in one release instance, we delay the start time until the latest first occurrence time of anomalous log events when all of them have a contrast weight of 0.

## 5. EXPERIMENT RESULTS

Release Instance ID	Anomalous Log Event Severity	# INFO	# WARN	# ERROR
14	WARN	69395	210	0
	WARN			
15	WARN	73053	239	1
	WARN			
16	WARN	72998	132	1
	WARN			
17	WARN	6732349	11054	9
18	WARN	4674690	2240	4
19	WARN	9240947	842460	9
20	WARN	11892	61	1

Table 5.2: Release instance information in test set, including 7 monitoring releases and 14 history releases from 6 hosts. Each row in the table represents one release instance, consisting of a monitoring release and two history releases in one host. Release instance ID is a manually assigned ID for each monitoring release and its two history releases. Each monitoring release might consist of a few anomalous log events, such as Release Instance 14. The number of logs of different severities are listed separately using the # symbol. For example, # INFO represents the number of INFO logs in the three releases of the corresponding release instance.

We utilize the total recall that considers both top 10 rankings of potential anomalous clusters and known clusters to evaluate the performance of pipeline on this data set. However, the drawback of this method is that the noise would be less because we move some logs from new log data to history log data by postponing the monitoring start time. In other words, the distributions of original data set and artificial data set are different. In this chapter, we construct a recurrent anomalies data set using 7 releases in the training set, including Release Instance 2, 3, 4, 7, 8, 9 and 11. The reason to choose them is that they still have more than half of unique log event sets remaining in new log data after postponing the starting time of the new log data. Another reason is that their anomalous log event have a severity of WARN or ERROR.

### 5.1.2 Labels

Developers would publish the anomalous logs during each release of the host in the public channel of Adyen. There are usually one or two anomalous log events in one release. We manually label anomalous log events and consider the rest as normal log events in a release.

Log events that the anomalous logs belong to are manually labeled as **anomalous log events**. Similarly, a log event set that consists of an anomalous log event is considered as an **anomalous event set**. Clusters are called **actual anomalous clusters** when they have an representative event set that is anomalous. The **potential anomalous clusters** mentioned before in Chapter 4 are the suspicious clusters and consist of only a few actual anomalous clusters. When a cluster in the ranking consists of an anomalous event set but has a normal

representative event set, it is not an actual anomalous cluster.

### 5.1.3 Benchmark

The parameters from [34] are used as the benchmark, where the authors use an agglomerative hierarchical clustering with Cosine distance. The authors first cluster log event sets in history log data and new log data respectively using the same **cluster distance** of 0.5. The distance between clusters is computed using their representative log event sets, which are the log event set in the cluster that has the smallest average distance to other members of the cluster. Later, the outlier score of each cluster in new log data is defined as its distance to its nearest cluster in history log data. Clusters with an outlier score greater than the **filter threshold** 0.5 are shown to developers for manual inspection. The filter threshold, first cluster distance, and anomaly cluster distance in [34] are all set to a constant value **0.5**.

The work of [34] clusters the new log data first then filters the anomalies. This has been shown not ideal in the preliminary experiment because the anomalous event sets get clustered with normal event sets, and the anomalous log event may not be representative of the cluster. In [34] long sequences would be processed directly in the clustering. This may lead to a low recall in Adyen logs as the long sequences tend to have a higher outlier score as they considered quite different from the short sequences. [34] also does not distinguish the different severity levels. To make a reasonable comparison, the benchmark in this chapter also goes through the cutting long sequences step and uses the same severity ratio value. In this light, the benchmark in this chapter is an improved version of [34] for Adyen logs.

### 5.1.4 Ranking functions

In our experiments two different ranking functions are used for the final output of anomalous clusters and known clusters. The first one is the outlier score computed by each anomaly detection algorithm in Equation 4.18. The second one is the average weight of the representative log event set of anomalous clusters or known clusters in Equation 4.19. Although earlier experiments show that the ranking function of average weight would be better than the outlier score, the outlier score ranking function is still used in experiments to reflect the actual performance of anomaly detection algorithms.

### 5.1.5 Packages

The experiments and visualization are completed using Python 2.7.10 with some Python packages:

- scipy (1.0.0)
- numpy (1.14.1)
- scikit-learn (0.19.1)
- seaborn (0.8.1)

IPLoM algorithm code is modified from Github repository <sup>1</sup> written by [22].

### 5.1.6 Evaluation Metric

In this chapter, we propose three evaluation metrics (*MAP*, recall and effort reduction) to evaluate the performance of the anomaly detection algorithms. As mentioned in 4.7, the final result returned by the pipeline would be an ordered list of **potential anomalous clusters** or **known clusters**. They are all obtained by clustering similar log event sets. The *effort reduction* metric is chosen to measure how well this final clustering of log event sets is working to merge the similar results. Because one of the important goals of the pipeline is to reduce the manual work for developers. The mean average precision (*MAP*) is chosen to evaluate the performance of the pipeline because *MAP* also considers the rankings of the result for precision. The *recall* metric is used to evaluate the ability of the pipeline to capture the real anomalies. A high recall is crucial for anomaly detection because the fraction of anomalies within the data is usually small. A low recall and a high accuracy indicate the pipeline is not able to detect anomalies.

As mentioned, only the top 10 are shown to developers, which means any log event cluster with a ranking lower than 10<sup>th</sup> would not be considered as anomalies. This is based on the idea that developers are generally like users who would be reluctant to go to the second page in the searching results of the search engine.

The *recall@10* is calculated using the **anomalous event sets**. An anomalous event set is only captured when it belongs to an **actual anomalous cluster** within top 10 rankings of potential anomalous clusters. To avoid duplicate rankings of log event sets within the same cluster, **potential anomalous clusters** instead of log event sets are used to calculate *MAP@10* of the anomaly detection. In the scope of the thesis, all *MAP* and average recall values are referred to *MAP@10* and *recall@10* unless mentioned explicitly otherwise. In summary, anomalous log events, anomalous event sets, or actual anomalous clusters are derived based on labels given by Adyen developers. The potential anomalous clusters are returned by the pipeline. We evaluate the performance of the pipeline by computing the number of captured actual anomalies and their rankings in the potential anomalous clusters.

#### Recall

The captured anomalous event sets are anomalous event sets that belong to an actual anomalous cluster within a certain ranking threshold. The recall is then defined as the fraction of captured anomalous event sets over the total number of anomalous event sets. The recall *R* is calculated as [2]

$$R(k) = \frac{|S(k) \cap G|}{|G|} \quad (5.1)$$

where *k* is a predefined ranking threshold, *S(k)* represents event sets from potential anomalous clusters within top *k*, and *G* is all anomalous event sets. For instance, the

---

<sup>1</sup><https://github.com/logpai/logparser>

$recall@10$  in our experiments would only consider the first 10 results in the rankings as captured anomalies.

### MAP: Mean Average Precision

An actual anomalous cluster is captured only when it has a ranking within a defined threshold. Precision is defined as the fraction of captured actual anomalous clusters over the total number of potential anomalous clusters. Given a predefined ranking threshold  $k$ , the potential anomalous cluster  $S(k)$  within top  $k$ , and all anomalous event sets  $G$ , precision  $P$  is calculated as [2]

$$P(k) = \frac{|S(k) \cap G|}{|S(k)|} \quad (5.2)$$

Average precision ( $AP$ ) is the mean of precision values of all captured actual anomalous clusters at certain position. Average precision is calculated as [9]

$$AP = \sum_{k=1}^n (R_k - R_{k-1}) P_k \quad (5.3)$$

where  $P_k$  is the precision,  $R_k$  is the recall, and  $k$  represents the  $k$ th position. Instead of treating the output results as unordered, the average precision also takes the rankings of the anomalies into consideration.

Mean Average Precision (MAP) is equal to the mean value of AP obtained from multiple queries, according to [9]. In our example, finding anomalies in each release instance is one query. Therefore, we calculate the average value of AP from release instances as the MAP. The formal definition of MAP is listed as follows [9]:

$$MAP = MEAN_{r \in R} AP(r) \quad (5.4)$$

where  $r$  represents one release instance, and  $R$  is a collection of release instances in our data set.

### Effort Reduction

When we detect multiple anomalous event sets, it is crucial to cluster them correctly to reduce the number of similar log event sets in the output in case they would mask other anomalies by occupying too many high ranks. The clustering of log event sets also prevents duplicate results in the final output. We borrow the concept of effort reduction from [34]. However, instead of considering all log events, we only examine anomalous log events for simplicity reasons. Given anomalous event sets  $E$ , and the collection of their corresponding clusters  $G$ , the effort reduction is computed as

$$Effort\ reduction = \frac{|E|}{|G|} \quad (5.5)$$

When there are multiple anomalous log events, the average *effort reduction* is calculated. The smaller the *effort reduction* is, the better the performance would be. There exists an

optimal minimum value for each release instance as the number of anomalous log events are a constant number for one release. This minimum *effort reduction* is obtained when all of the anomalous event sets are put into one cluster.

Besides simplicity, another reason that effort reduction is not calculated for normal log event sets is that it is hard to measure objectively whether they should be clustered together or not. Anomalous log events in the anomalous event sets are usually, at least in our data set, the templates with the highest weight computed by Equation 4.7. This means the corresponding anomalous event sets should be cluster together because the anomalous event sets share the same highest weight templates.

### 5.1.7 Distance Metrics

Some popular metrics are compared to choose the suitable metric. In the following definitions,  $D$  represents for distance, and  $X, Y$  represent for vectors with  $N$  dimensions. Hamming distance and Jaccard distance are generally calculated for binary vectors, the weighted versions are used instead for better comparison.

1. Cosine distance

$$D_{Cosine} = 1 - \frac{X \cdot Y}{\|X\|_2 \|Y\|_2} = 1 - \frac{\sum_{i=1}^n x_i y_i}{\sqrt{\sum_{i=1}^n x_i^2} \sqrt{\sum_{i=1}^n y_i^2}} \quad (5.6)$$

2. Manhattan distance or city block distance

$$D_{Manhattan} = d_1(\mathbf{X}, \mathbf{Y}) = \|\mathbf{X} - \mathbf{Y}\|_1 = \sum_{i=1}^n |x_i - y_i| \quad (5.7)$$

3. Hamming distance

$$D_{Hamming} = \frac{\sum_{i=1}^n |x_i - y_i|}{n} \quad (5.8)$$

4. Euclidean distance

$$D_{Euclidean} = \|\mathbf{X} - \mathbf{Y}\|_2 \quad (5.9)$$

5. Jaccard distance

$$D_{Jaccard} = \frac{|X \cap Y|}{|X \cup Y|} = \frac{|X \cap Y|}{|X| + |Y| - |X \cap Y|} \quad (5.10)$$

6. Pearson distance

$$D_{Pearson} = 1 - \frac{(X - \bar{X}) \cdot (Y - \bar{Y})}{\|X - \bar{X}\|_2 \cdot \|Y - \bar{Y}\|_2} = 1 - \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}} \quad (5.11)$$

## 5.2 Feature Extraction Results

### 5.2.1 Log Abstraction Methods

Since the primary research questions focus on the modeling the logs in the release, we are not going to compare different log abstracting methods. Because it requires much manual work to inspect if each log is abstracted into the correct corresponding template. The *Iterative Partitioning Log Mining* (IPLoM) described in [36][37] is chosen because its efficiency and accuracy stated in [22] where multiple message abstracting methods are compared with different log data sets. IPLoM is a heuristic algorithm and runs at  $O(n)$  time complexity, where  $n$  is the number of logs.

IPLoM is quite accurate in Adyen date set. A template is correctly abstracted if it does not consist of logs from other templates. An average of **92.1%** log templates are correctly abstracted, according to our manual inspection with logs from five releases. There are mainly two causes for the uncorrected abstracted templates. The first is that log messages are too short, making the IPLoM falsely finding the mapping among unrelated pairs. The second is that some log messages are of JSON format, it is hard to abstract the templates in the JSON format. Because they are quite different from other message strings written by developers. Although the keys of the JSON format are extracted beforehand and are used input of abstraction, the performance is still not ideal because the order of keys and number of keys can affect the result of IPLoM.

### The Reconciling Clustering Step of Similar Templates

In this step, similar templates are clustered using the threshold of **0.4** based on the similarity calculated in Equation 4.2. The recall of the reconciling step is used to measure the performance of this reconciling step. According to our manual inspection of logs from 5 releases, the reconciling step reaches an average recall of 88.8% by measuring how many similar templates are correctly clustered.

Two ratios are used to measure the performance of IPLoM and its reconciling step with reducing the size of log data for analysis. We use the training set to calculate two ratios. On average, the number of log events is only 0.4% of the number of raw logs. Moreover, after reconciling step, the number of template clusters are further reduced to 74.2% of the number of the log templates.

$$\frac{\# \text{ Log Templates}}{\# \text{ Raw Logs}} = 0.00402$$

$$\frac{\# \text{ Template clusters}}{\# \text{ Log Templates}} = 0.742$$

### 5.2.2 The Weight Distribution of Log Templates

The *idf* distribution from Release Instance 1 is shown in Figure 5.1. When a log template appears in fewer task IDs, it would have a higher *idf* weight. This figure clearly shows that

## 5. EXPERIMENT RESULTS

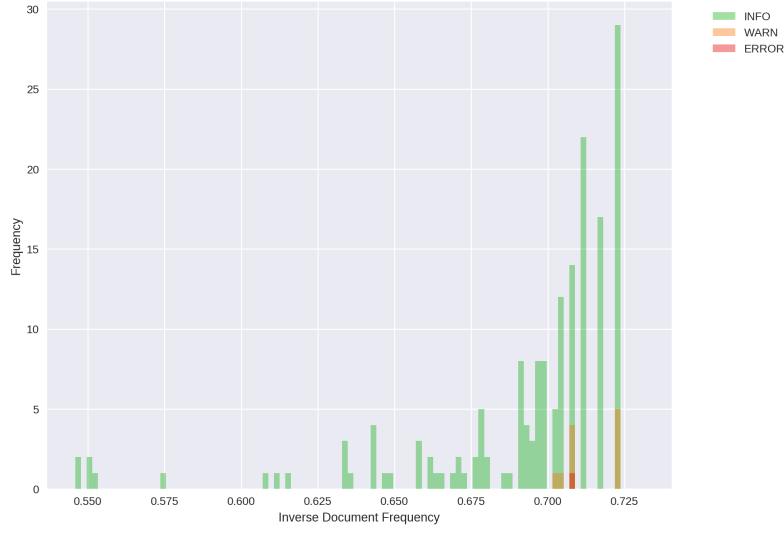
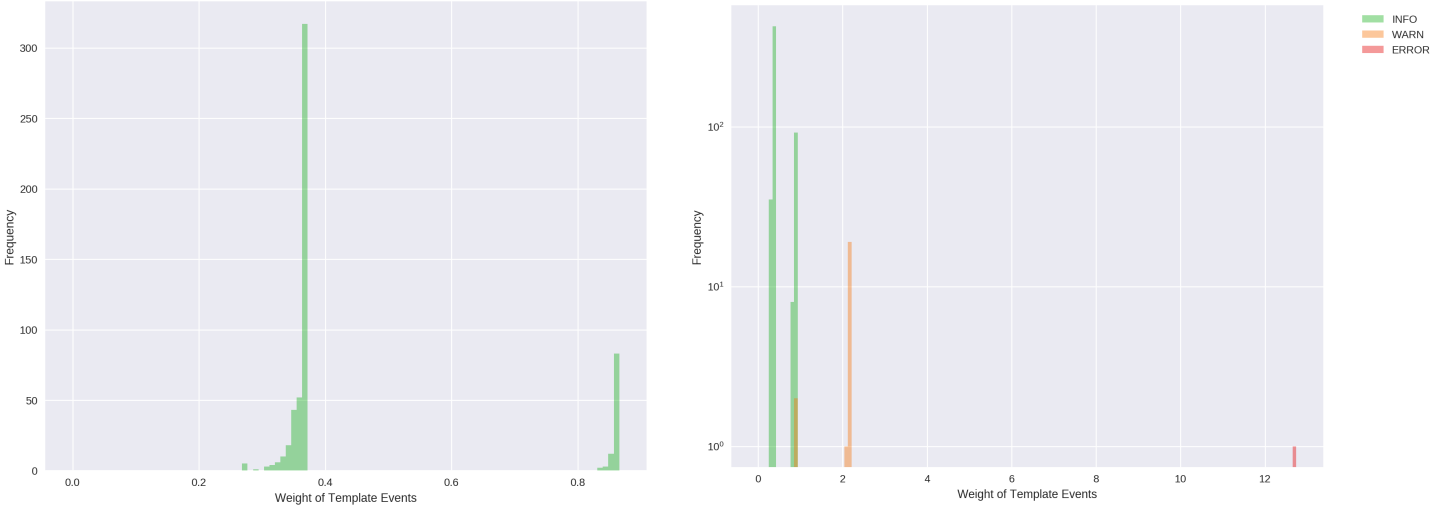


Figure 5.1: The distribution normalized *idf* weight of log templates from Release Instance 1. The x-axis represents the value of *idf* weight, y-axis is the frequency of log events with a certain *idf* weight. Green columns represent INFO logs, yellow columns represent WARN logs, and red columns represent ERROR logs.



(a) Distribution of overall weights of INFO log templates

(b) Distribution of overall weights of all log templates

Figure 5.2: The distribution of overall weights of log templates from Release Instance 1. The severity ratio is set to 10. The left figure shows only the INFO log templates weight, and the right figure shows all three type of log templates. Green columns represent INFO logs, yellow columns represent WARN logs, and red columns represent ERROR logs.



WARN and ERROR templates are less frequent than most INFO templates and therefore have a higher *idf* weight.

The overall weight of each template calculated by Equation 4.7 is shown in Figure 5.2. The severity ratio is set to 10 here as an example. In Figure 5.2b, the WARN and ERROR templates have a higher overall weight than INFO templates because they have a higher severity weight. In Figure 5.2a, the overall weights of INFO templates are separated into two groups. One group has an overall weight lower than 0.5 and another group has an overall weight higher than 0.5. This difference is caused by the contrast weights. The group on the left with a smaller overall weight consists of log templates that have been seen before the release. While the group on the right side of the figure consists of new log templates that have a contrast weight of 1. The splitting of two groups caused by contrast weights can also be found in the WARN templates as shown in Figure 5.2b. In Figure 5.2b, if a lower severity ratio is assigned in Equation 4.7, templates with different severities would have a closer overall weight.

### 5.2.3 The Distributions of Length and Time Duration of Sequences

The time duration of one log sequence equals the difference between the timestamp of the latest log and that of the earliest log in the sequence. The distribution of time duration of log sequences from Release Instance 1 is shown in Figure 5.3a. The length of a log sequence is equal to the number of raw logs in the sequence. The distribution of length of log sequences from Release Instance 1 is shown in Figure 5.3b. We only consider log sequences formed by PSP references rather than thread IDs here, which allows us to compute the threshold with sequences with normal length.

As stated before, we use the 99-percentile length value of log sequences that formed by PSP references as the threshold to cut the long sequences. However, sometimes the 99-percentile threshold is still too long, such as 64 in Release Instance 1. To solve this, we also utilize threshold of time duration. We find the 99-percentile values of the time duration of log sequences have a PSP reference are all less than 2. We cut the long sequences using the two thresholds. If a sequence lasts longer than the time threshold, it would be cut into short sequences based on the timestamp. If the sequence is not cut according to the time duration threshold, we check if the sequence is longer than the length threshold. It would still be cut into short sequences if its length exceeds the threshold.

## 5.3 Anomaly Detection Algorithms

This section mainly discusses the choice of different parameters of anomaly detection algorithms, including clustering, nearest neighbor, and LOF. Corresponding experiment results are used to support the choice by using the three evaluation metrics from Section 5.1.6.

At first, six distance metrics are compared by how well they cluster the bug event sets together, after which three distance metrics are chosen for the other experiments. Later, clustering is used to find the optimal values for two general parameters, filter threshold and anomaly cluster distance which are shared by all three anomaly detection algorithms.

## 5. EXPERIMENT RESULTS

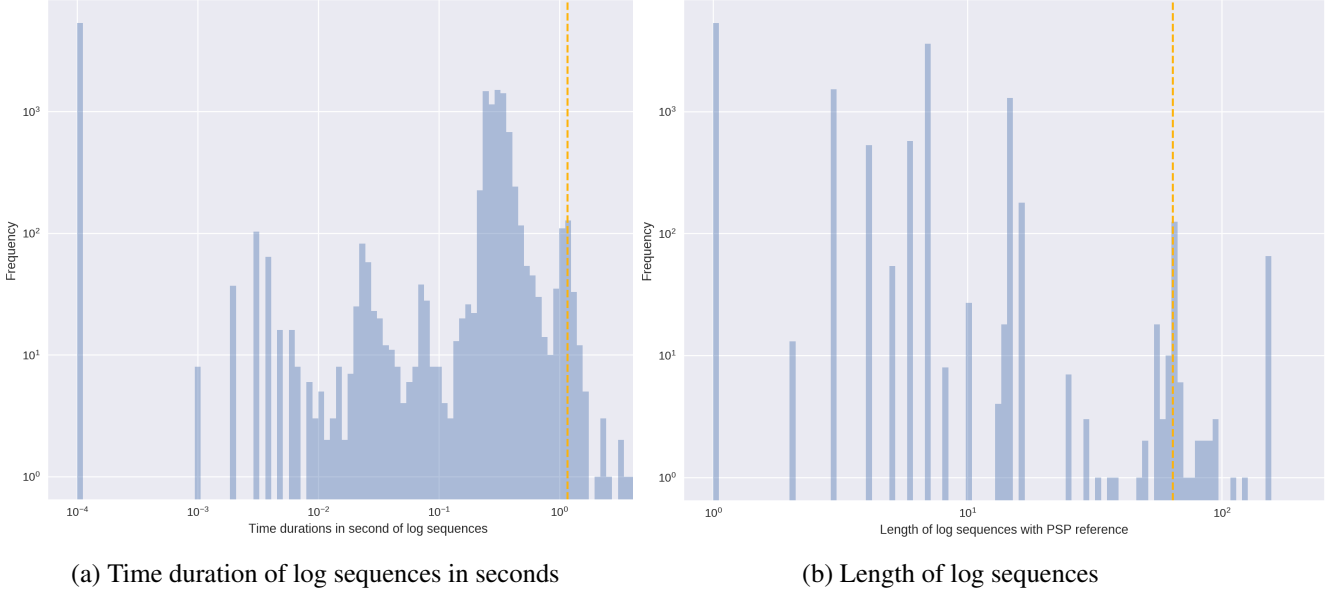


Figure 5.3: The distributions of length and time duration of log templates weights from Release Instance 1. The distribution only include the log sequences formed with a PSP reference. The amber dashed lines represent 99-percentile values in each distribution.

Moreover, the influence of severity ratio in Equation 4.5 is also shown for each anomaly detection.

Besides the shared parameters in the pipeline, the specific parameters for each anomaly detection algorithm are also tested, such as the first clustering distance in the hierarchical clustering and the value of  $k$  in nearest neighbor or LOF. We use the training set from Table 5.1 to get the optimal parameters in this section.

### 5.3.1 Effort Reduction of Distance Metrics

A proper distance metric is essential for the performance of proximity algorithms. Hierarchical clustering is used as the anomaly detection algorithm in this experiment to compute the *effort reduction* of each distance. The values of average *effort reduction* of releases in the training set with different distance metrics are compared under different cluster thresholds (Equation 4.12); the result is shown in Figure 5.4. The x-axis is the cluster percentile; the y-axis represents the value of effort reduction that is calculated by dividing the number of actual anomalous clusters with the number of anomalous event sets.

Unsurprisingly, Euclidean distance does not work well in the outlier clustering because it generally suffers from the curse of dimensionality [3]. Hamming distance and Manhattan distance also do not perform well. Usually anomalous event sets are not clustered together unless a very high *cluster distance* is chosen for these three distance metrics. The Pearson distance and Cosine distance give the best results among different distances by having

the lowest cluster number. When *cluster distance* increases, more anomalous event sets are clustered together; the effort reduction of Euclidean distance, Hamming distance, Manhattan distance, and Jaccard distance decrease. The performance of Pearson and Cosine, however, is not affected significantly by choice of *cluster threshold*.

The distribution of outlier scores of Release Instance 1 computed by hierarchical clustering (Equation 4.10) is shown in Figure 5.5 to get a better idea of why some metrics fail to cluster similar log sequences. The outlier scores from hierarchical clustering are the distance between new data points with the corresponding nearest cluster. The yellow dashed lines represent for 12.5%, 25%, 50%, 75% and 87.5% percentile, the columns represent the frequency in the column, the short blue lines on the bottom is the actual appearance of the distance, and the blue curves are the kernel density estimation. The Cosine distance (5.5a) and Pearson distance (5.5b) almost have the same distribution, while Manhattan distance (5.5f) and Hamming distance (5.5e) are more similarly. Jaccard distance (5.5c), Cosine distance and Pearson distance have a flatter distribution and a maximum value of 1, while the other three distance metrics have a big spike on the left.

## Conclusion

Therefore Cosine, Pearson, and Jaccard distance metrics are chosen for further experiments.

### 5.3.2 Impact of Filter Threshold

In this section we again use a hierarchical clustering algorithm in anomaly detection to choose the optimal value for filter threshold in Equation 4.11. Any data point with an outlier score that exceeds the **filter threshold** would be considered as potential anomalies. A strict filter threshold can neglect actual anomalies. On the other hand, a low value of filter threshold would generate too many false positives.

Before choosing the suitable value for filter threshold, we plot outlier scores of anomalous event sets in each release instance from the training set in Figure 5.6. Almost all anomalous event sets have an outlier score which is greater than 0.5 when using Cosine, Pearson, or Jaccard. Data points with a low outlier score shown in this figure are anomalous event sets of Release Instance 3 and 8. They have low outlier scores because they have similar log event sets before the release. INFO anomalies of Release Instance 5 and 6 still have a high outlier score in three distance metrics; in Cosine distance, for example, the outlier scores are 0.6212 and 0.5545 respectively.

In order to determine the exact values of **filter distance** and **cluster distance** calculated from the percentiles, percentile distances are plotted in Figure 5.7. As shown in the Figure 5.7, when a large percentile is chosen, the actual distance value calculated from different hosts may vary greatly.

## Conclusion

In order not to miss the potential anomaly sequences, a filter percentile of 50 is used, and the corresponding filter distance is calculated from Equation 4.11. The filter distance can be up to 0.76 with the percentile of 50 for Jaccard, which may lead to the decreasing of

## 5. EXPERIMENT RESULTS

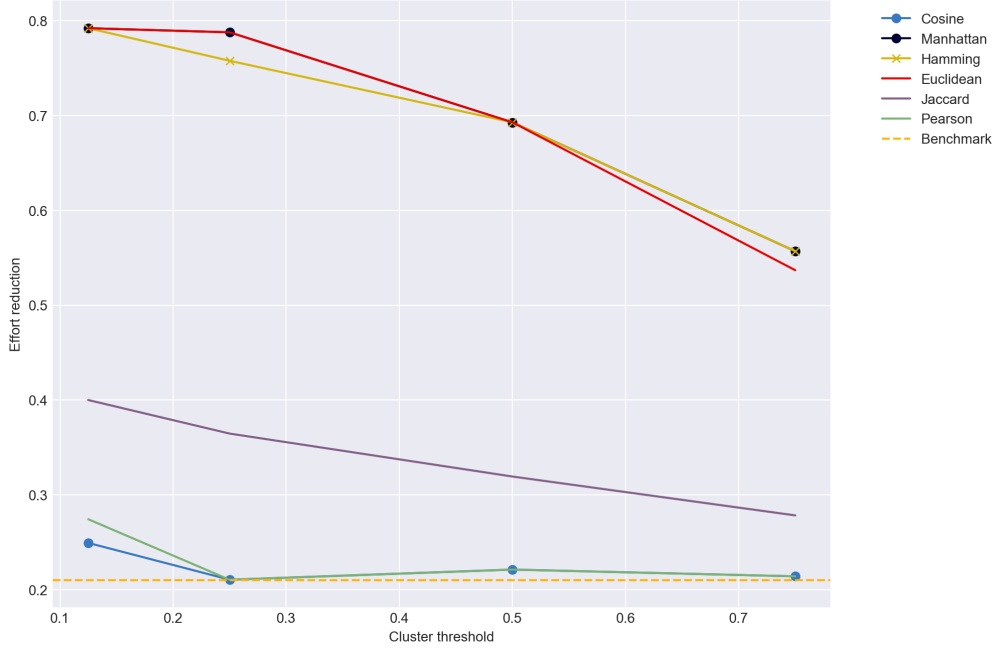


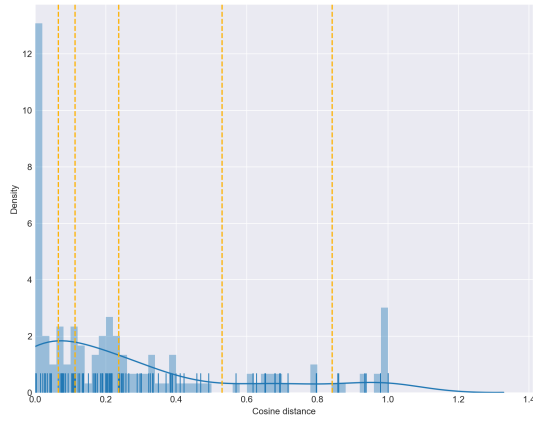
Figure 5.4: The average effort reduction of releases in the training set with different distance metrics under different cluster percentiles. The x-axis is the cluster percentile in Equation 4.12. The y-axis represents the value of effort reduction that is calculated by dividing the number of actual anomalous clusters with the number of anomalous event sets. We use lines with markers to show metrics that are overlapped by others.

Recall @ 10. A high filter threshold means a strict condition on choosing anomalies and may overlook actual anomalies. Thus the filter threshold is necessary to compare with 0.50. We choose the smaller value between the two as shown in Equation 5.12, which is used for later other experiments.

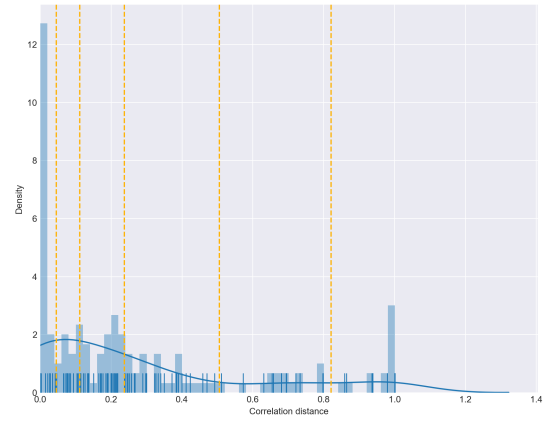
$$\text{Filter threshold} = \min(0.5, \text{Percentile}(\text{non-zero OA}, 50)) \quad (5.12)$$

### 5.3.3 Impact of Anomaly Cluster Distance

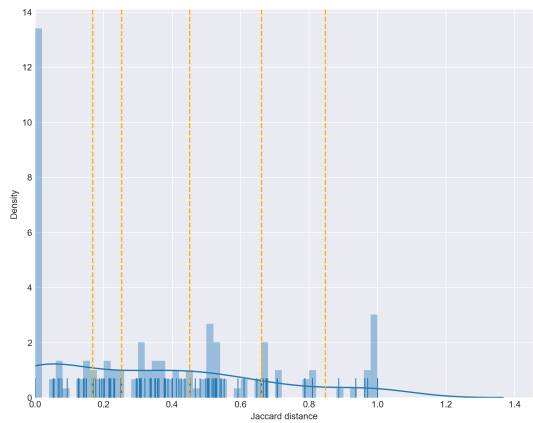
Besides the filter threshold, another important parameter in the pipeline is the anomaly cluster distance. The **anomaly cluster distance** is set as the distance threshold of hierarchical clustering to cluster the suspicious event sets that are detected by the algorithms. If this cluster distance is too small, it would prevent the suspicious event sets from clustering together. On the other hand, if this distance is set too large, clusters can get big and mask the potential anomalies because they may not be the representative event sets in the anomaly cluster.



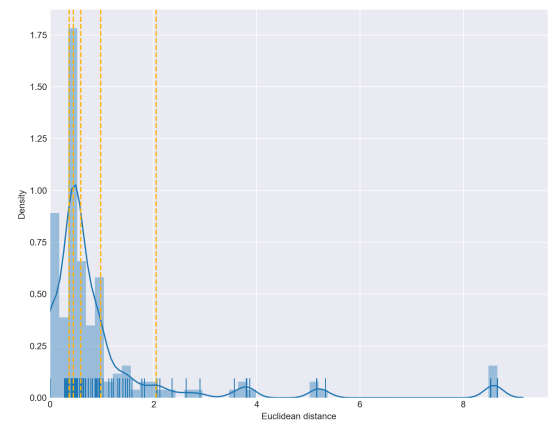
(a) Cosine distance



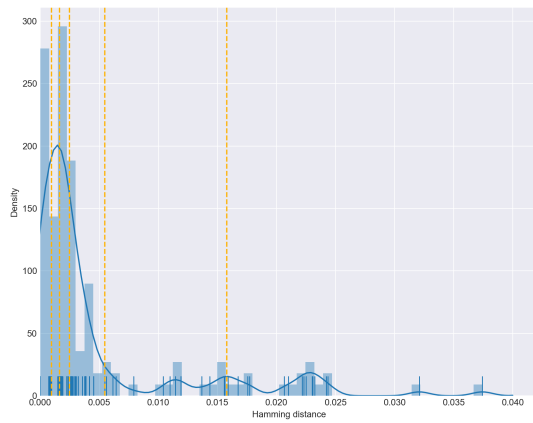
(b) Pearson distance



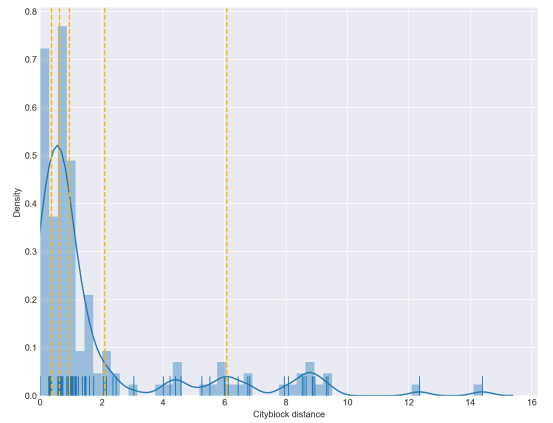
(c) Jaccard distance



(d) Euclidean distance



(e) Hamming distance



(f) Manhattan distance

Figure 5.5: The distribution of outlier scores of Release Instance 1 computed by hierarchical clustering (Equation 4.10). The yellow dashed lines represent for 12.5%, 25%, 50%, 75% and 87.5% percentile values of the outlier scores, the columns represent the frequency in the column, the short blue lines on the bottom is the actual appearance of the distance, and the blue curves are the kernel density estimation.

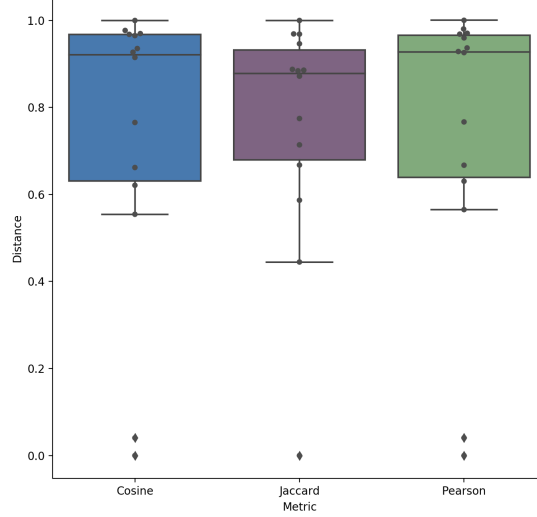


Figure 5.6: Boxplot of outlier scores of anomalous event sets in each release instance from the training set, which is computed by hierarchical clustering in different metrics. Distance values of three distance metrics are separated into three groups.

We choose to use hierarchical clustering for anomaly detection again in this experiment. The performance of pipeline using different anomaly cluster distances, both on the original data set and INFO anomalies data set, is plotted in Figure 5.8 and Figure 5.9. In these types of experiments, we utilize MAP and average recall to evaluate the performance of the pipeline under different values of a specific parameter. The MAP is computed as the average AP of each release instance using the rankings of actual anomalous clusters in the data set. Similarly, the value of recall is also calculated using the average recall of anomalous log event sets in release instances. It is worth noting that we only consider the **potential anomalous clusters** here, ignoring known clusters.

Usually we would use Cosine, Jaccard, and Pearson distance metrics along with two ranking functions in Equation 4.18 and Equation 4.19. In total, we have six combinations of distance metrics and ranking functions. For example in Figure 5.8, the legend on the right represents the six combinations: the colors represent the corresponding distance metric, and the dots represent average ranking function (Equation 4.19), and triangles are outlier score ranking function (Equation 4.18). A “Cosine weight” legend entry means using Cosine distance metric for the distance and the average weight for computing the rankings of potential anomalous clusters. On the other hand, “Cosine distance” means that we use the outlier score as the ranking function, even though in LOF the outlier score is not the distance. The purpose of keeping outlier score as the ranking function is to evaluate the actual performance of anomaly detection algorithms.

As shown in the Figure 5.8 and Figure 5.9, the performance of using average weight

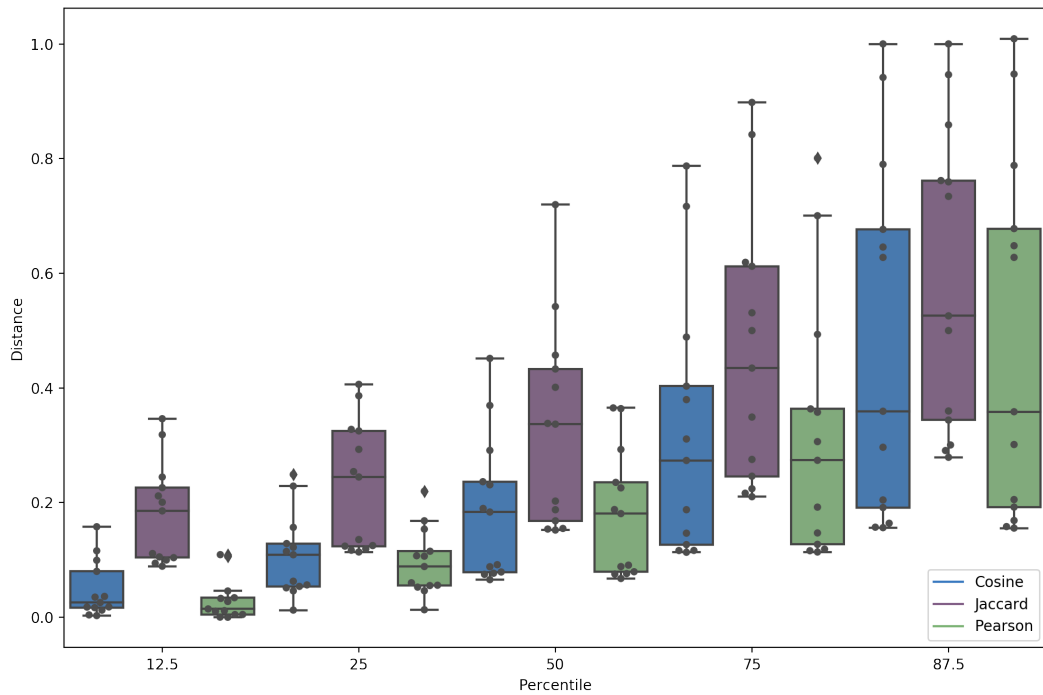


Figure 5.7: Boxplot of percentile values of outlier score of new log data in the training set, which is computed by hierarchical clustering in different metrics. The x-axis is the percentile and y-axis is the distance value computed by the corresponding percentile. In each percentile value, distance values of three distance metrics are separated into three groups.

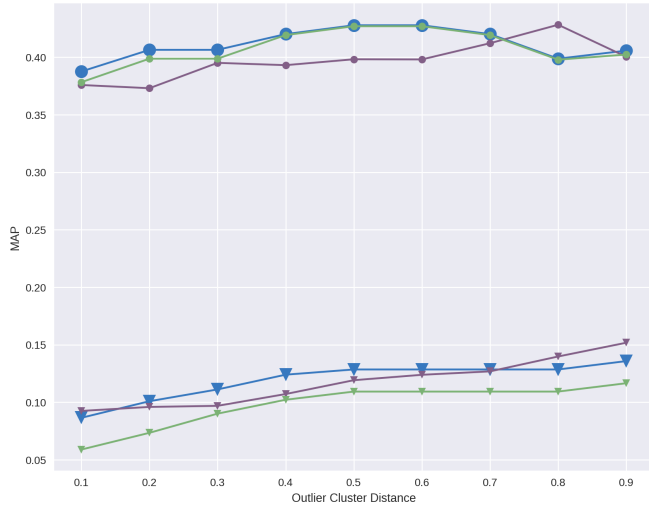
scores as the ranking function is almost always better than using outlier score, considering both MAP and average recall. This is mainly because new log events are added in every code release, and a high outlier score represents the novelty of the log event sets. For example, log event sets that consists of only new INFO logs may be of high rank when considering the distance score. However, when the average weight is used as the ranking function, the logs with higher rankings are those event sets with more critical log events, for example, new ERROR and WARN logs.

Usually, average weight ranking works. When there are a lot of new WARN and ERROR logs detected, nevertheless, log event sets with only INFO events may get neglected because they do not have a high average weight due to the low severity score. There is a compromise between getting severer logs and getting more INFO errors.

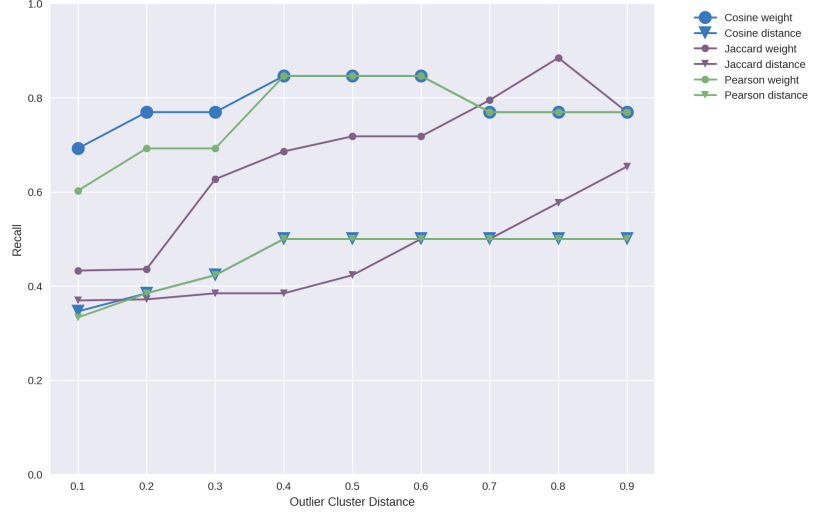
## Conclusion

In Figure 5.8 and Figure 5.9, the best performance of the clustering is also obtained by setting the cluster distance directly to **0.4**, considering three distance metrics.

## 5. EXPERIMENT RESULTS

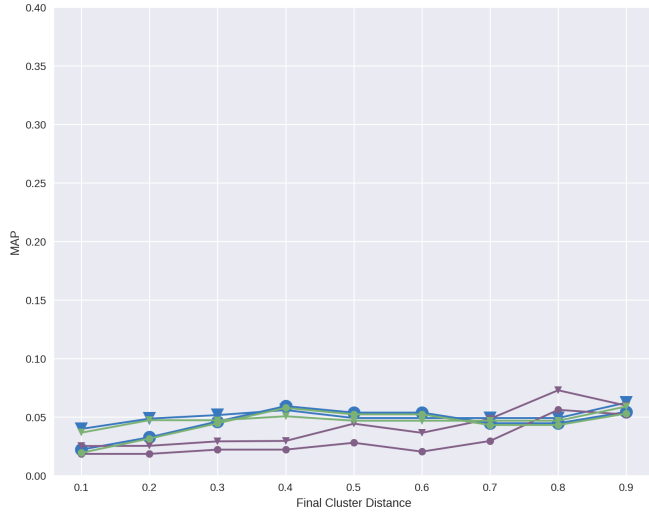


(a) MAP of distance metrics

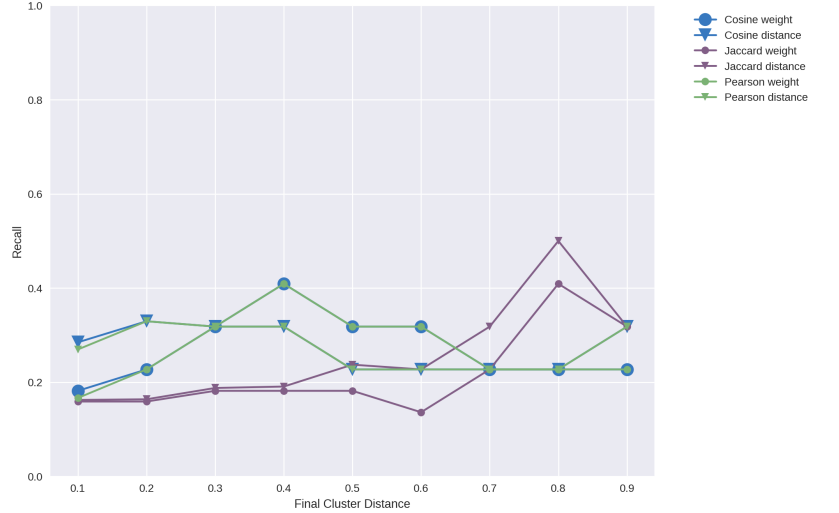


(b) Average recall of distance metrics

Figure 5.8: MAP and average recall of clustering for original data set in the training set using different *anomaly cluster distances*, where x-axis is the anomaly cluster distance, and y-axis is the value of MAP or recall. We use lines with dots to represent the average weight ranking function (Equation 4.19) and lines with triangles to represent the outlier score ranking function (Equation 4.18).



(a) MAP



(b) Average recall

Figure 5.9: MAP and average recall of clustering for INFO anomalies data set using different *anomaly cluster distances*, where x-axis is the anomaly cluster distance, and y-axis is the value of MAP or recall. We use lines with dots to represent the average weight ranking function (Equation 4.19) and lines with triangles to represent the outlier score ranking function (Equation 4.18).



### 5.3.4 Impact of First Cluster Distance

When using the hierarchical clustering for anomaly detection, we would cluster history log data and use **first cluster distance** as the distance threshold. The benchmark from [34] uses a first cluster distance of 0.5 for Cosine distance. The value of first cluster distance would also affect the performance of the pipeline. If we choose first cluster distance strictly, the fewer clusters we obtain in the clustering of history log data. If this distance is large enough, eventually only one cluster is obtained, and all new data point would have the same outlier scores. If this distance is small enough, the number of clusters would equal the number of history log data points, generating the same result as the 1-nearest neighbor distance algorithm. The reasonable choice is to choose also **0.4** because it is better to use the same value as the **anomaly cluster distance**. In order to see if 0.4 is also an optimal choice for the first cluster distance, several experiments are done below.

The ratio between the number of history clusters and the number of history event sets is computed for each release as in Figure 5.12. We use this figure to visualize the influence which first cluster distance exerts on the clustering of the history log data. In this figure the curved lines with triangles are the average value of ratios in every release instance, and each distance metric is assigned with one color. The dots are ratios from different release instances. The colored zones are the standard deviation of the ratios with respect to release instances under different metrics. This ratio is a number in the range of  $[0, 1]$ . When the number of clusters gets closer to the number of event sets, the ratio becomes closer to 1. In Figure 5.12, the average ratio value lines of Cosine and Pearson are almost overlapping. On the other hand, Jaccard has more clusters as it has a higher ratio compared to Cosine and Pearson.

The performance of the pipeline using different values of first cluster distance is shown in Figure 5.10. Moreover, the performance of the pipeline to detect INFO anomalies is shown in Figure 5.11. MAP and average recall in Figure 5.10 for the original data set are quite robust to *first cluster distance* when the value of first cluster distance is smaller than 0.5. The MAP of the outlier score ranking function and the recall of both ranking functions decrease as the clustering increase. As expected, MAP and average recall of *INFO anomalies data set* are lower than that of *original data set*, but they are still robust to the choice of first cluster distance.

### Conclusion

The recommended value for the first cluster distance is **0.4** for Adyen data set because the recall is almost the highest for both original and INFO anomalies data set.

### 5.3.5 Impact of Severity Ratio

One of the assumptions of the anomaly detection is that the higher the severity of the log event, the more likely it is anomalous. Log events of different severities should be assigned with different weights. Different severity ratios in Equation 4.5 would affect both the filtering and clustering steps described in Section 4.6. When we use a high severity ratio, log

## 5. EXPERIMENT RESULTS

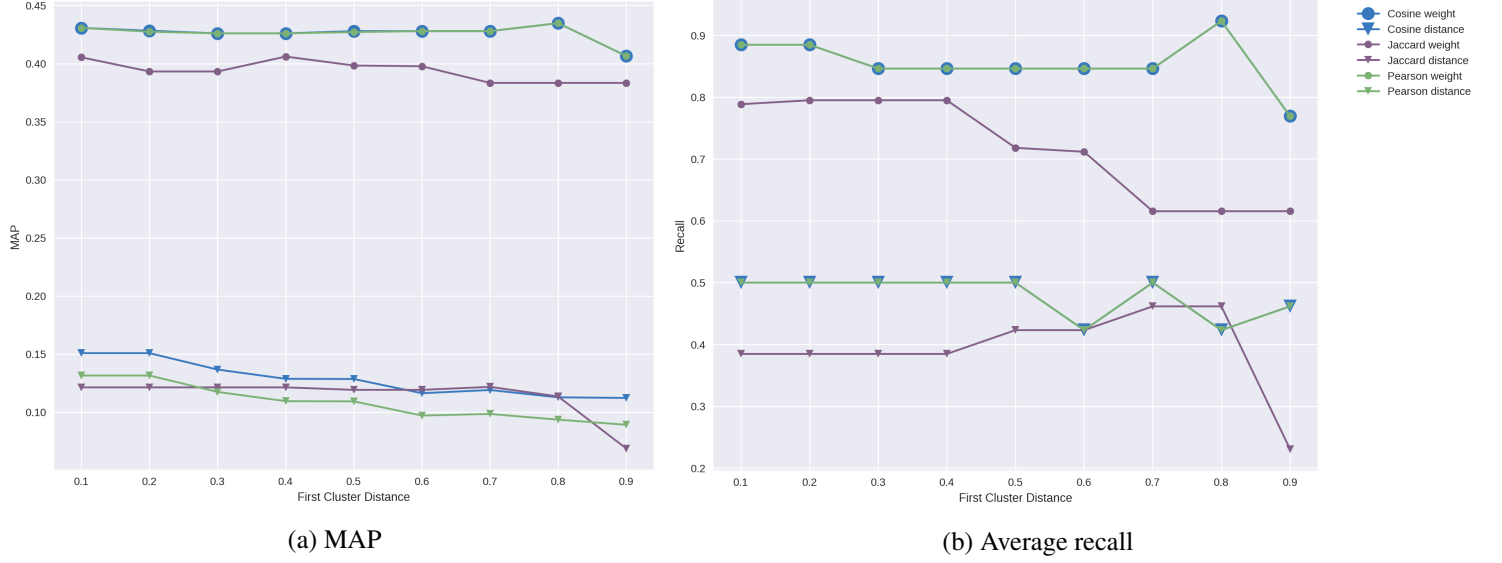


Figure 5.10: MAP and average recall of clustering for original data set in the training set using different *first cluster distances*, where x-axis is the first cluster distance, and y-axis is the value of MAP or recall. We use lines with dots to represent the average weight ranking function (Equation 4.19) and lines with triangles to represent the outlier score ranking function (Equation 4.18).

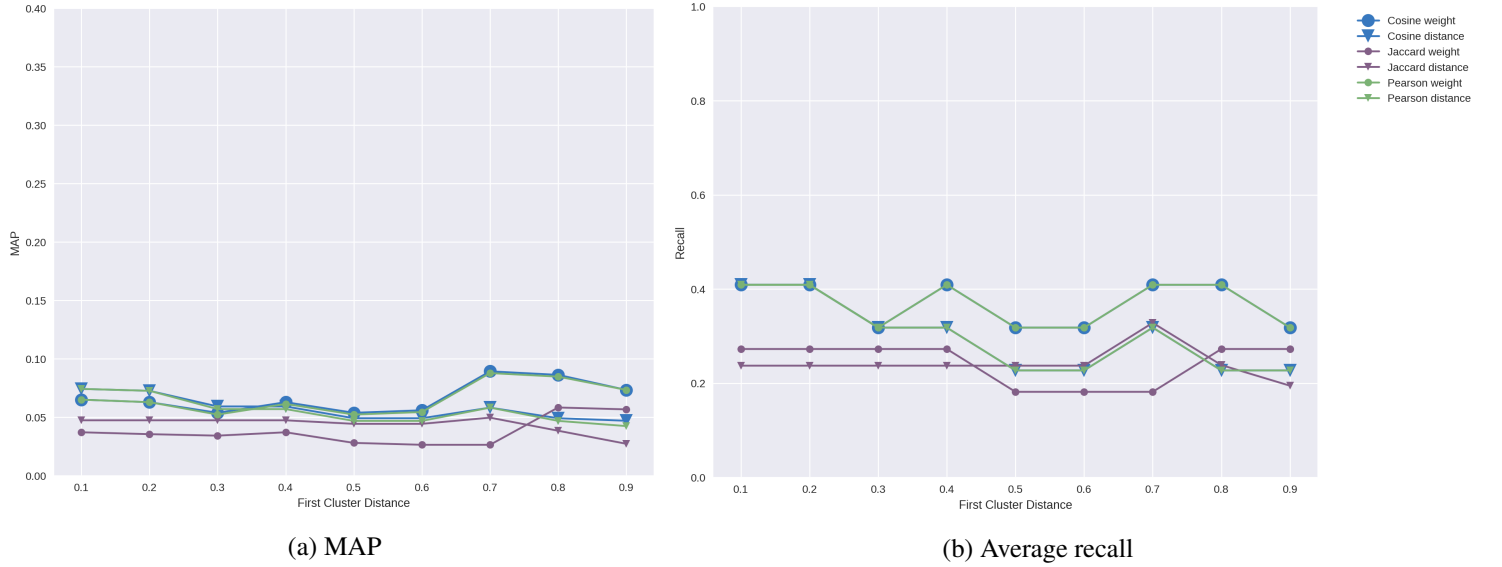


Figure 5.11: MAP and average recall of clustering for INFO anomalies data set in the training set using different *first cluster distances*, where x-axis is the first cluster distance, and y-axis is the value of MAP or recall. We use lines with dots to represent the average weight ranking function (Equation 4.19) and lines with triangles to represent the outlier score ranking function (Equation 4.18).

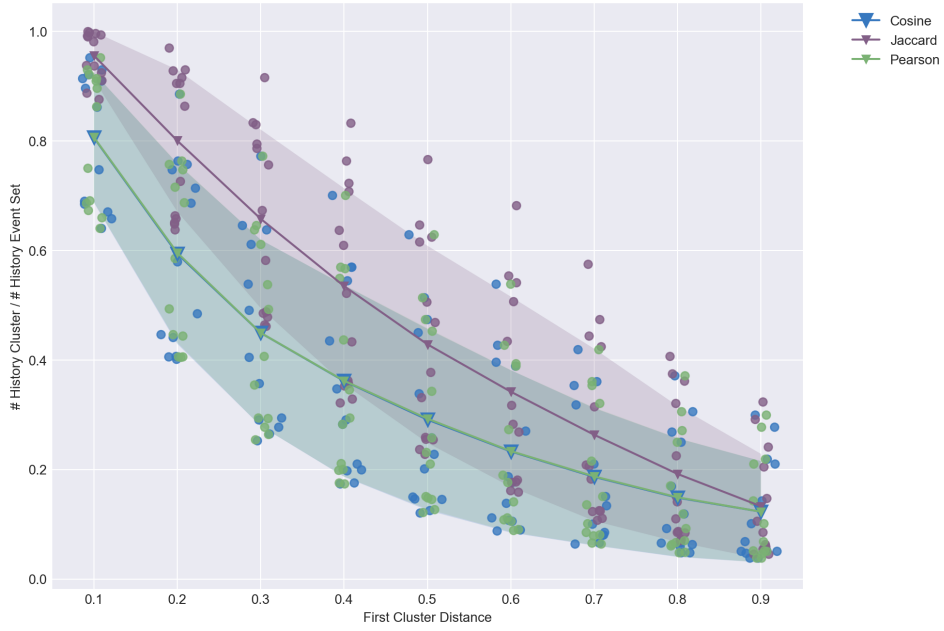


Figure 5.12: Ratio of history cluster number and history log event set using different *first cluster distances* in the training set, where the x-axis represents the first cluster distance, the y-axis represents the value of the ratio. The curved lines with triangles are the average value of ratios in every release instance. The dots are ratios from different releases. The colored zones are the standard deviation of the ratios with respect to release instances under different metrics.

event sets that consist of WARN or ERROR events would get a higher ranking in the final output. On the other hand, the pipeline tends to oversee INFO anomalies.

We choose to use hierarchical clustering for anomaly detection again in this experiment. To find the suitable severity ratio, we evaluate the performance of the pipeline using different values of severity ratio. The result is shown in Figure 5.13. MAP and average recall also increase when severity ratio increases, as shown in Figure 5.13 because the majority of anomalous event sets have a severity of WARN or ERROR.

To test how severity ratio would affect the capability of the pipeline to capture INFO anomalies, we also use the INFO anomalies data set. The result is shown in Figure 5.14. As expected, the MAP of both ranking functions decreases in Figure 5.14a. The outlier score ranking outperforms the average weight ranking for INFO anomalies although the average weight is more robust to the changing of severity ratio. However, recall of using average weight is slightly worse than using outlier distance in Figure 5.14b. When considering different distance metrics, Jaccard performs far from ideal compared with the other two metrics.

The severity ratio is set to 4 because MAP and average recall are already the quite high for original data set at this value. So there is no need to increase the severity ratio further. Although the MAP would not be the highest for INFO anomalies when severity

## 5. EXPERIMENT RESULTS

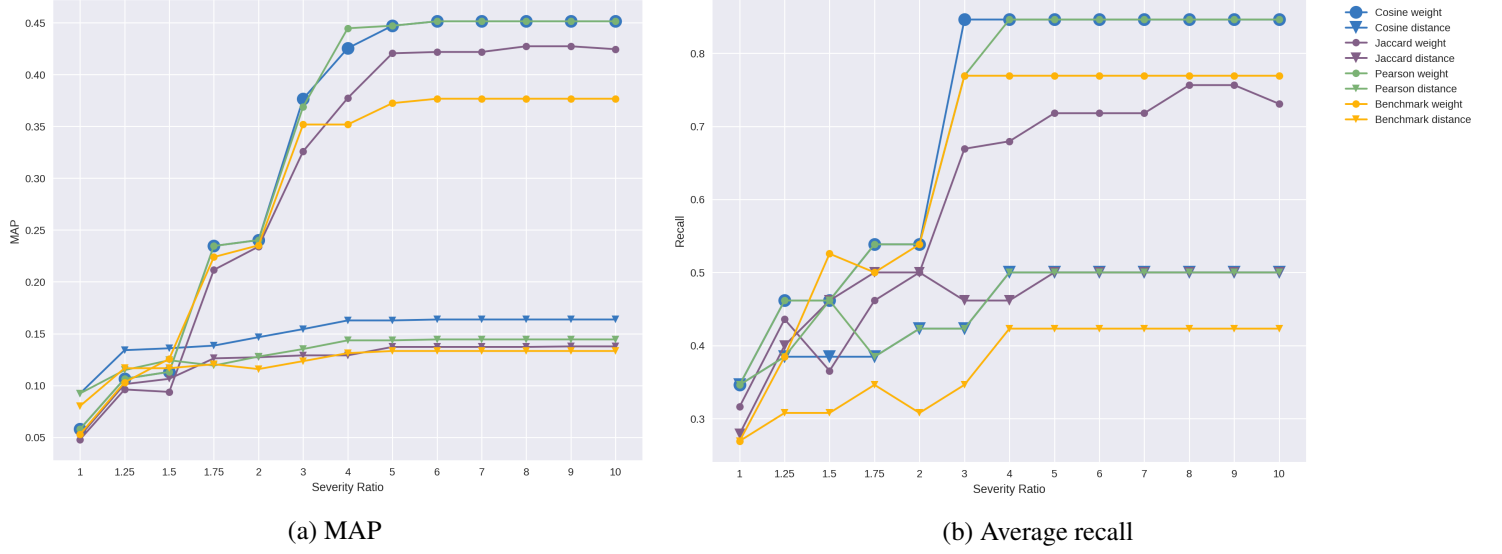


Figure 5.13: MAP and average recall of clustering for original data set using different *severity ratios*, where x-axis is the severity ratio, and y-axis is the value of MAP or recall. We use lines with dots to represent the average weight ranking function (Equation 4.19) and lines with triangles to represent the outlier score ranking function (Equation 4.18).

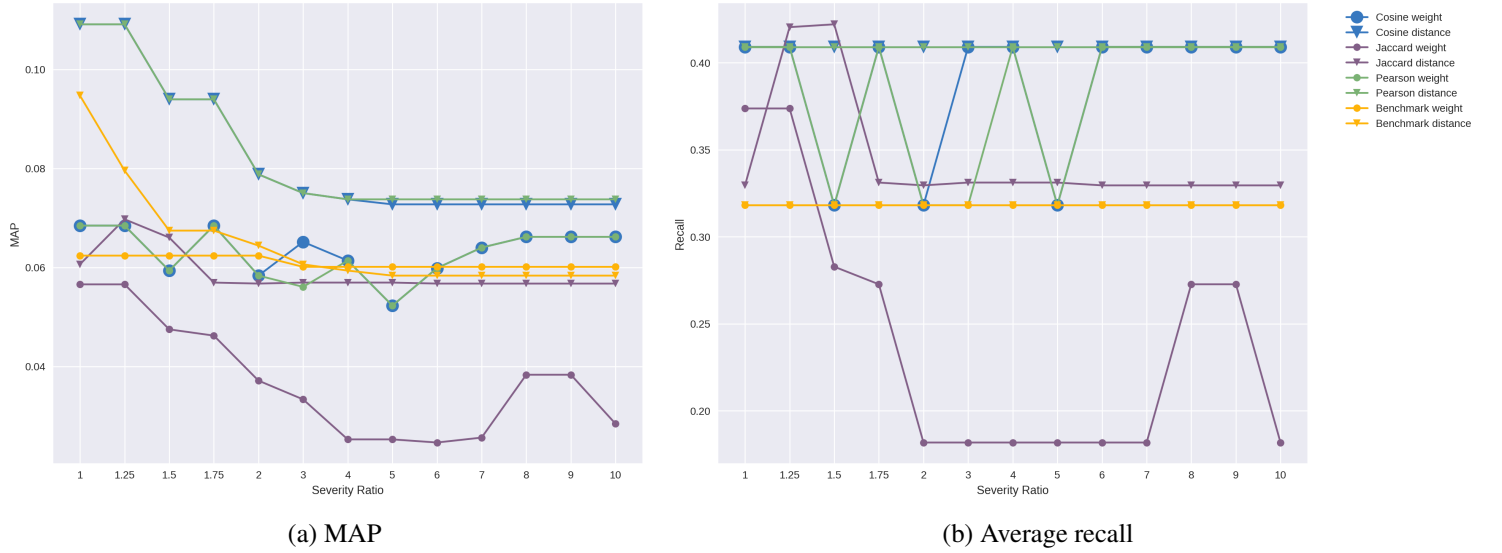


Figure 5.14: MAP and average recall of clustering for INFO anomalies data set using different *severity ratios*, where x-axis is the severity ratio, and y-axis is the value of MAP or recall. We use lines with dots to represent the average weight ranking function (Equation 4.19) and lines with triangles to represent the outlier score ranking function (Equation 4.18).

ratio equals 4, average recall is always at the range from 0.32 to 0.41 for Cosine and Pearson distances despite the value of severity ratio. This means the pipeline can still capture the INFO anomalies, but they are in a lower rank.

### Conclusion

Thus the severity ratio is set to **4** for Equation 4.5 because of the reasons mentioned above, the final calculation equation is shown in Equation 5.13. This severity ratio value might not be suitable for other anomaly detection algorithms such as nearest neighbor distance or LOF. We are going to explore the impact of severity ratio for these algorithms separately in the following sections.

$$w_{severity}(i) = \begin{cases} 1, & \text{if } i \text{ is INFO} \\ 4, & \text{if } i \text{ is WARN} \\ 16, & \text{if } i \text{ is ERROR} \end{cases} \quad (5.13)$$

#### 5.3.6 Detecting Recurrent Anomalies in Known Clusters

Logs generated after the release are more likely to be anomalies. However, it is likely that there are already anomalies generated before the release because the current monitoring host is affected by the release process of other hosts. Unfortunately, according to the assumption stated in 3.4.2, the logs happened before the release before despite the high severity would be considered as normal data, which may make the algorithms fail to capture the potential anomalies that happen right before the release. We show that the pipeline is still robust against recurrent anomalies with the help of known clusters, which is a complementary step we design to capture recurrent anomalies.

In this experiment, we focus on **recurrent anomalous log events** that appear in both history log data and new log data. We try to understand how the ranking of known clusters would help the pipeline to deal with the recurrent anomalous log events. The starting time of log data are changing accordingly with the first occurrence time of the labeled anomaly log events so that they are first seen right before the release, which would lead to contrast weight of 0 in Equation 4.4 and a rather low outlier score. By doing so, we would get a **recurrent anomalies data set** from the original data set as already discussed in Section 5.1.1.

We choose to use hierarchical clustering for anomaly detection again in this experiment. Solely recall is used to evaluate the performance in Table 5.3 because some anomalies are captured in both rankings of potential anomalous clusters and known clusters. We use average recall and average overall recall to evaluate the performance. Recall is computed similarly as before by computing the rankings of only potential anomalous clusters. On the other hand, **overall recall** considers both the rankings of potential anomalous clusters and known clusters. It is worthwhile to mention that the overall recall top 10 in two rankings of clusters, which would be in total 20 results that would be shown to developers.

As we can see in Table 5.3, the average overall recall is much higher of than the average recall of the pipeline, indicating most anomalies are captured in known clusters. The com-

## 5. EXPERIMENT RESULTS

plementary step indeed improves the robustness of the pipeline when dealing with recurrent anomalies.

Ranking Function	Metric	Average Recall	Average Overall Recall
Average Weight	Cosine	0.0714	0.9286
	Jaccard	0.1119	0.6952
	Pearson	0.0714	0.9286
Outlier Score	Cosine	0	0.8571
	Jaccard	0	0.6119
	Pearson	0	0.8571

Table 5.3: Average recall and average overall recall of the pipeline with recurrent anomalies data set, using hierarchical clustering for anomaly detection. The recurrent anomalies data set consists of 7 release instances from the training set. The ranking function in the first column only indicate the ranking function used by potential anomalous clusters. The average recall is computed using the rankings of **potential anomalous clusters**. The average overall recall is computed using the rankings of both **potential anomalous clusters** and **known clusters**.

### 5.3.7 Nearest Neighbor Distance

Clustering can be viewed as the 1-nearest neighbor using the representative log event sets. Nearest neighbor distance would perform in a more robust way when compared with multiple nearest neighbor distance to get a better granularity of how far new event sets are actually deviate from the history log data.

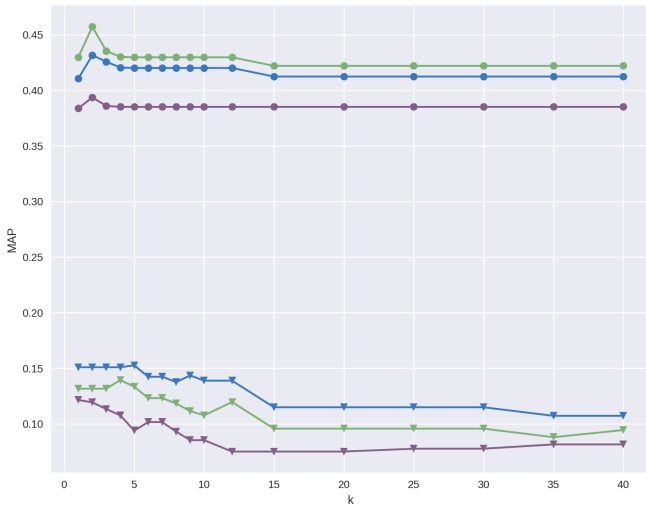
#### Impact of $k$

The  $MAP@10$  and  $recall@10$  of the nearest neighbor algorithm with different  $k$  values are plotted in Figure 5.15. The performance using average weight are quite robust to the choice of  $k$ . While the performance of the using outlier distance could be affect greatly by the value of  $k$ .

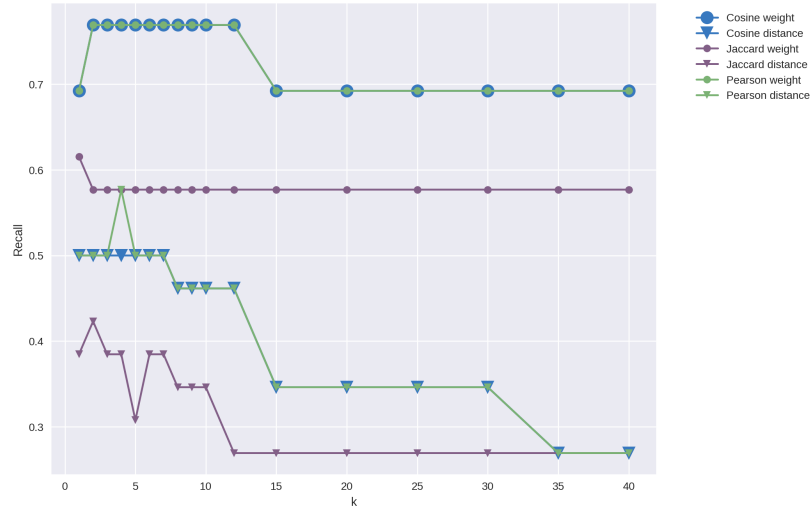
As shown in Equation 4.13, outlier scores which are computed using nearest neighbor distance are equal to the average distance of  $k$  nearest neighbor. Though in Figure 5.17 outlier scores of anomalous event sets increase as the number of neighbors increase, the outlier scores of all new event sets shown in Figure 5.18 also increase. This makes the recall of using outlier distance decrease a lot because when the number of considered neighbors increase, most new log event sets would be considered as far away from the old event sets.

The recommended  $k$  value is **5** as the recall for both ranking functions using Cosine and Pearson metrics are nearly optimal.

### 5.3. Anomaly Detection Algorithms

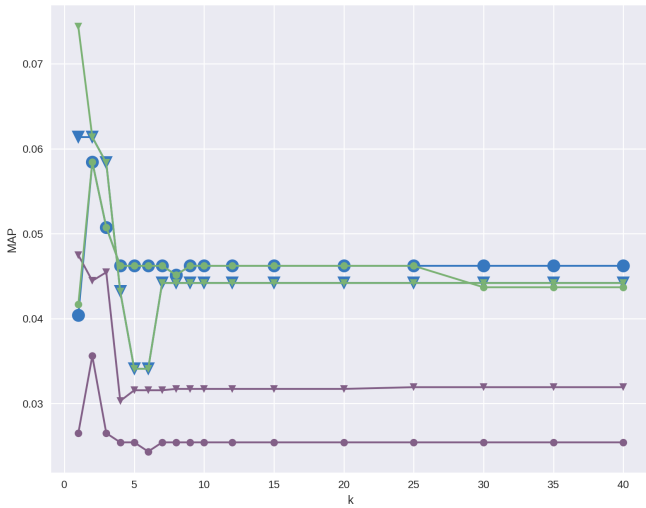


(a) MAP

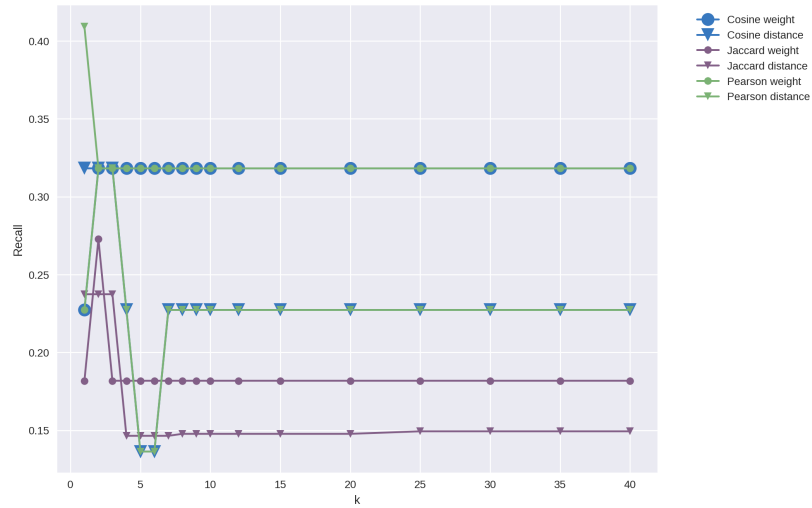


(b) Average recall

Figure 5.15: MAP and average recall of nearest neighbor distance for original data set using different  $k$  values (severity ratio = 6), where  $x$ -axis is  $k$ , and  $y$ -axis is the value of MAP or recall. We use lines with dots to represent the average weight ranking function (Equation 4.19) and lines with triangles to represent the outlier score ranking function (Equation 4.18).



(a) MAP



(b) Average recall

Figure 5.16: MAP and average recall of nearest neighbor algorithm for INFO anomalies under different  $k$  values (severity ratio = 6), where  $x$ -axis is  $k$ , and  $y$ -axis is the value of MAP or recall. We use lines with dots to represent the average weight ranking function (Equation 4.19) and lines with triangles to represent the outlier score ranking function (Equation 4.18).

## 5. EXPERIMENT RESULTS

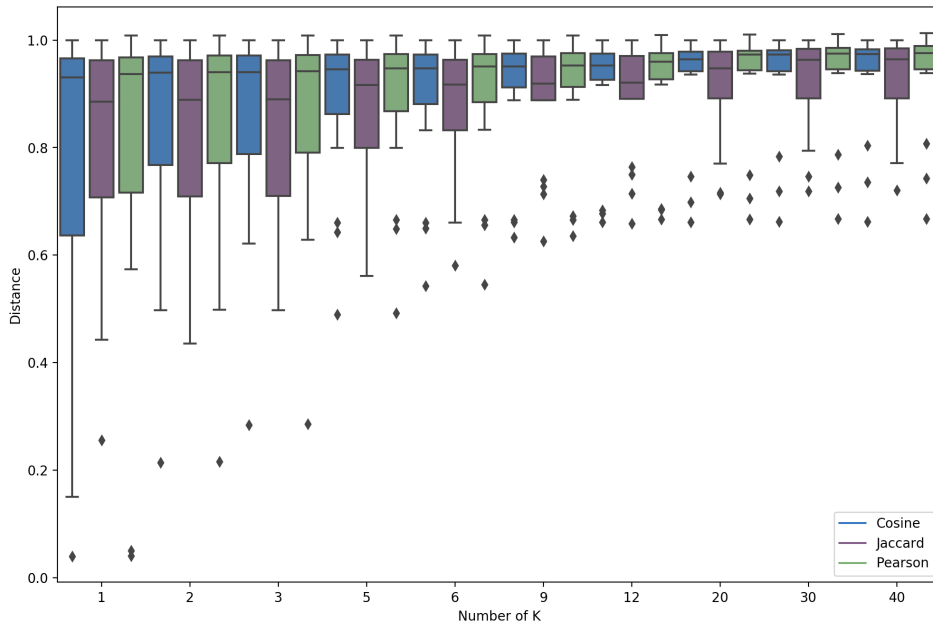


Figure 5.17: Distance of anomalous event sets with different  $k$  values, where x-axis is  $k$ , and y-axis is the outlier score of anomalous event sets.

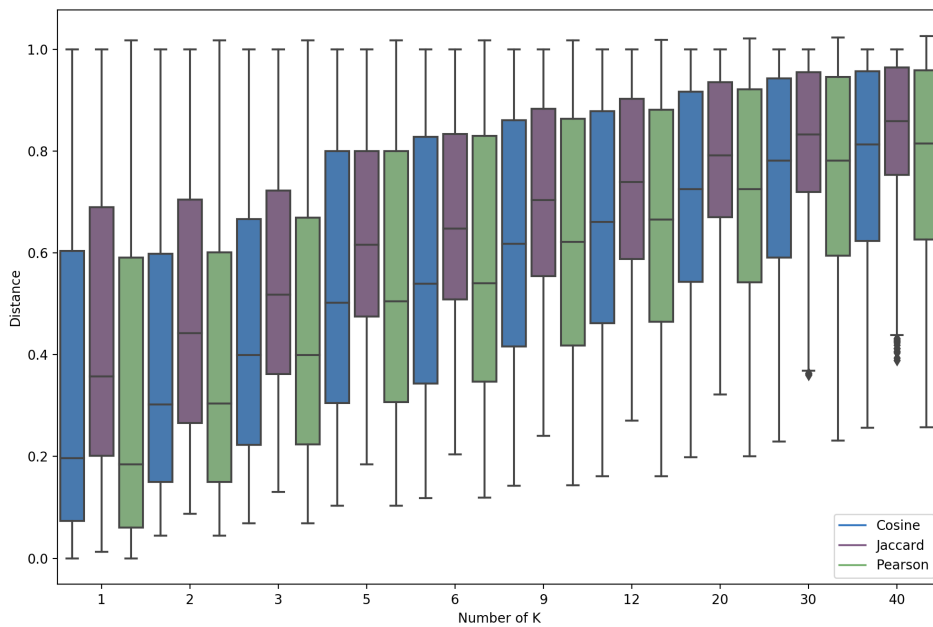


Figure 5.18: Distance of new log event sets with different  $k$ , where x-axis is  $k$ , and y-axis is the outlier score of anomalous event sets.

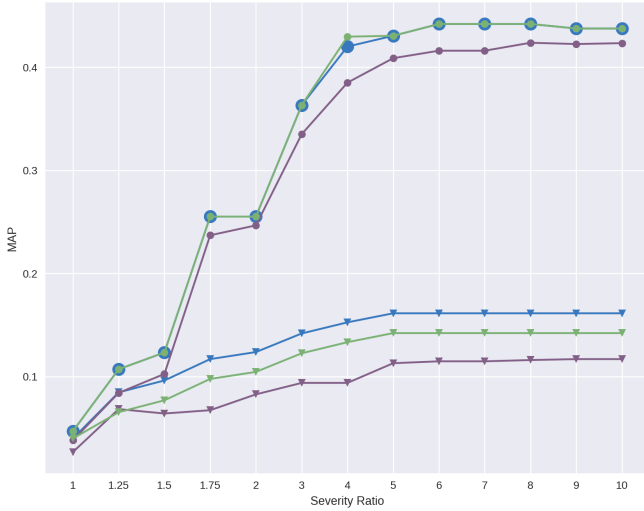


### Severity Ratio

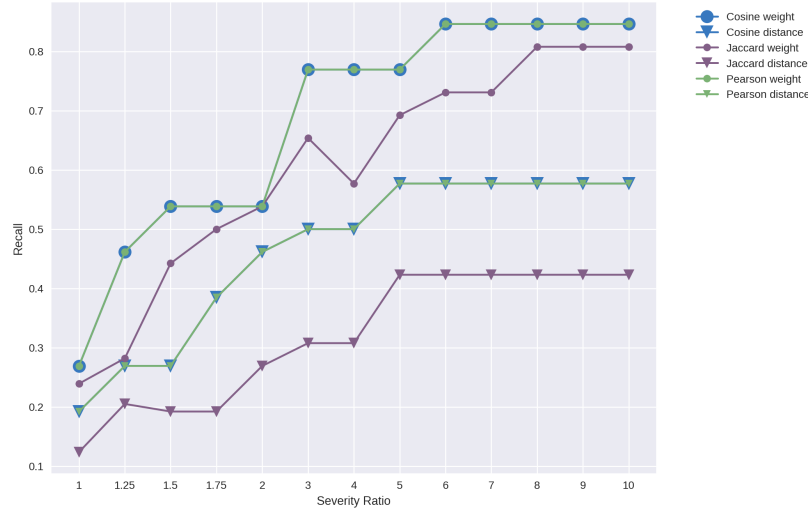
Unlike clustering algorithm, the nearest neighbor need severity ratio greater than **6** to get an maximum recall as shown in Figure 5.19. As most anomalous events are considered as higher severity, nearest neighbor algorithm use 5 nearest neighbors data points to calculate the distance threshold, would make new log sequences have a higher outlier score despite their severity.

For nearest neighbor, the severity ratio of **6** is chosen instead.

$$w_{severity}(i) = \begin{cases} 1, & \text{if } i \text{ is INFO} \\ 6, & \text{if } i \text{ is WARN} \\ 36, & \text{if } i \text{ is ERROR} \end{cases} \quad (5.14)$$



(a) MAP with different severity ratios



(b) Average recall with different severity ratios

Figure 5.19: MAP and average recall of nearest neighbor algorithm with different severity ratios ( $k = 5$ ), where x-axis is the severity ratio, and y-axis is the value of MAP or recall. We use lines with dots to represent the average weight ranking function (Equation 4.19) and lines with triangles to represent the outlier score ranking function (Equation 4.18).

### 5.3.8 Local Outlier Factor

As the LOF value is a bit different than the distance, it makes no sense to compare with 0.5 and get the minimum filter threshold as in Equation 5.12.

$$\text{Filter threshold} = \text{Percentile}(\text{non-zero OA}, 50) \quad (5.15)$$

## 5. EXPERIMENT RESULTS

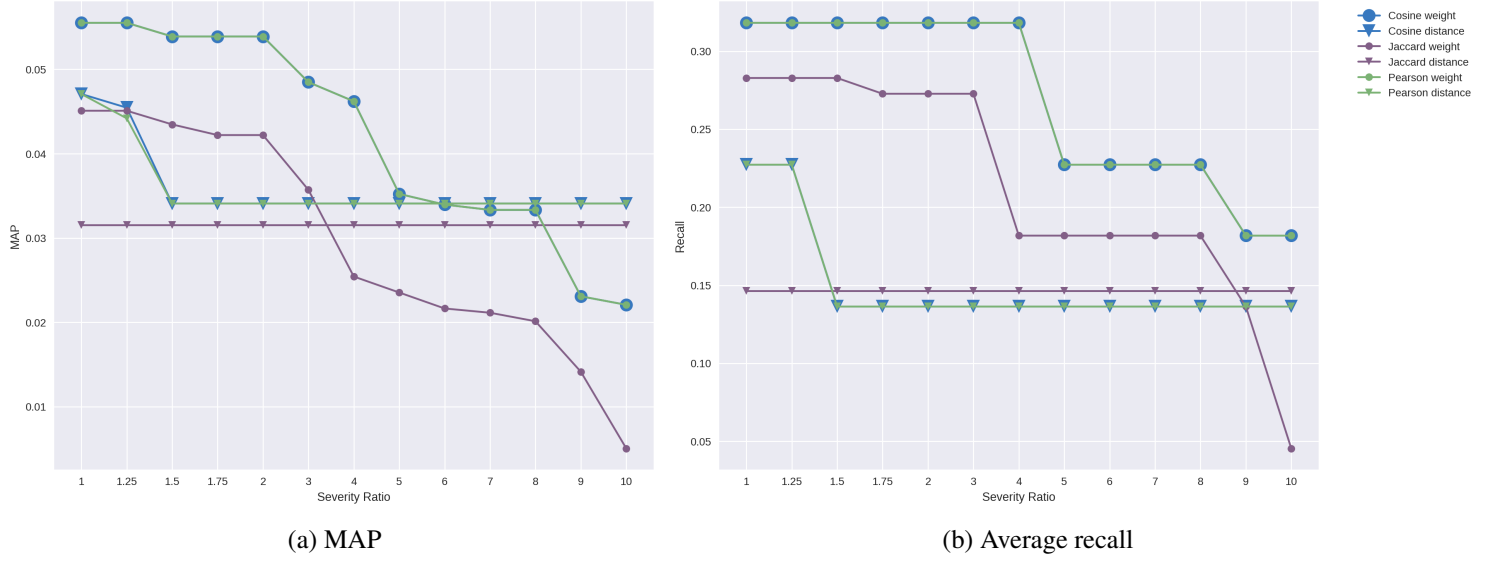


Figure 5.20: MAP and average recall of nearest neighbor to capture INFO Anomalies under different severity ratios ( $k = 5$ ), where x-axis is the severity ratio, and y-axis is the value of MAP or recall. We use lines with dots to represent the average weight ranking function (Equation 4.19) and lines with triangles to represent the outlier score ranking function (Equation 4.18).

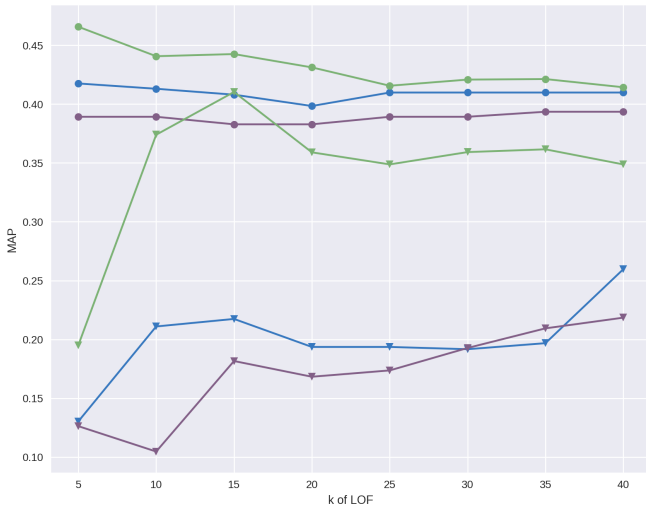
### Impact of $k$

As stated in [2] the value of  $k$  exerts more significant influence on the performance of LOF than on nearest neighbor distance algorithm. According to Figure 5.21, the *MAP* and recall values would be influenced more by the value of  $k$ . The larger the  $k$ , the more nearest neighbor the algorithm needs to take into consideration. If the value of  $k$  is too low, for example, lower than 10, then the LOF would fluctuate dramatically as the  $k$  changes according to [7]. The recall of the Cosine and Pearson distance metrics get maximum when the  $k$  equals 10, although the MAP using the outlier score ranking increase as the  $k$  change from 10 to 15. The recommended value for  $k$  is **15** for higher recall and more robust performance.

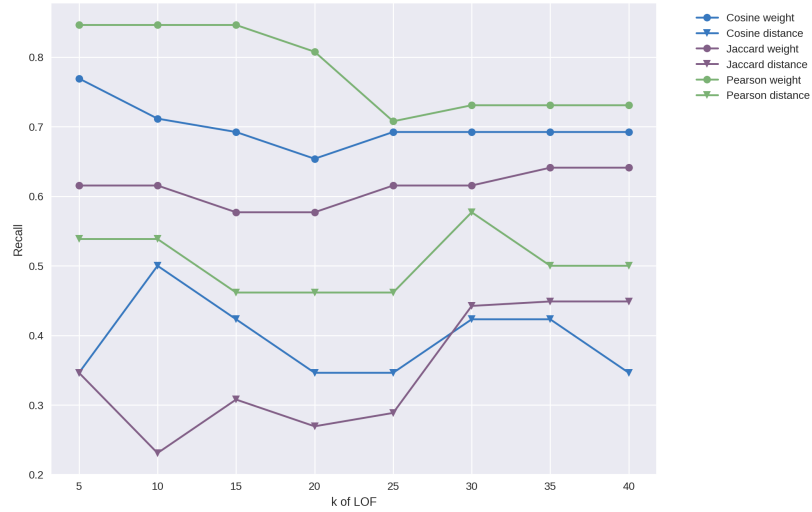
### Severity Ratio

Different severity ratio would affect the performance of LOF as well as shown in Figure 5.23. While the performance of LOF after chaining all anomalies to INFO verbosity is shown in Figure 5.24. The severity ratio of **4** is chosen for LOF as the same as clustering in Equation 5.13.

### 5.3. Anomaly Detection Algorithms

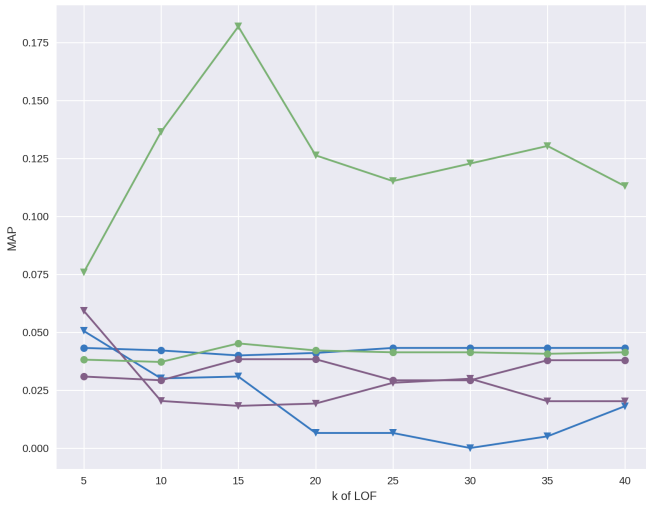


(a) MAP

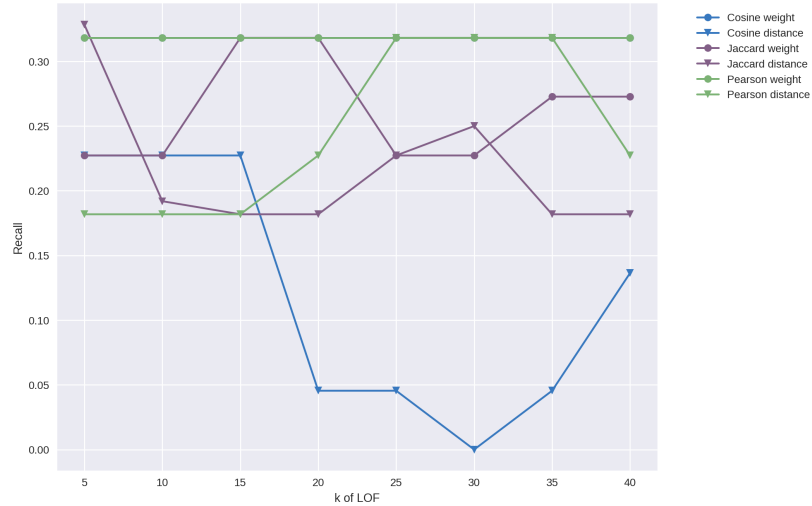


(b) Average recall

Figure 5.21: MAP and average recall of LOF for original data set under different  $k$  (severity ratio = 4), where x-axis is  $k$ , and y-axis is the value of MAP or recall. We use lines with dots to represent the average weight ranking function (Equation 4.19) and lines with triangles to represent the outlier score ranking function (Equation 4.18).



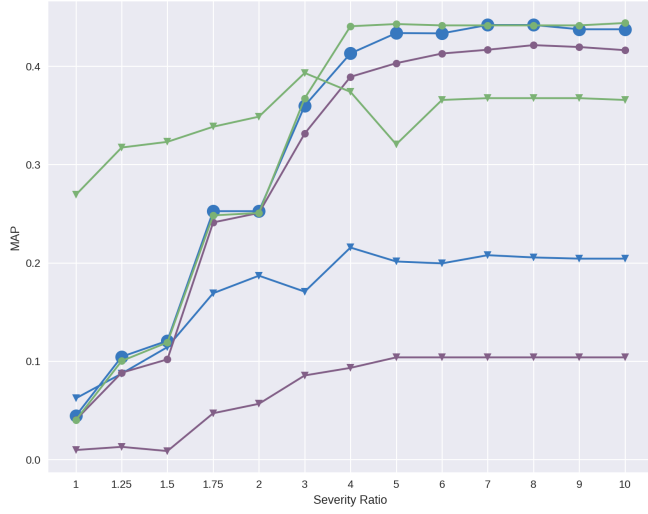
(a) MAP



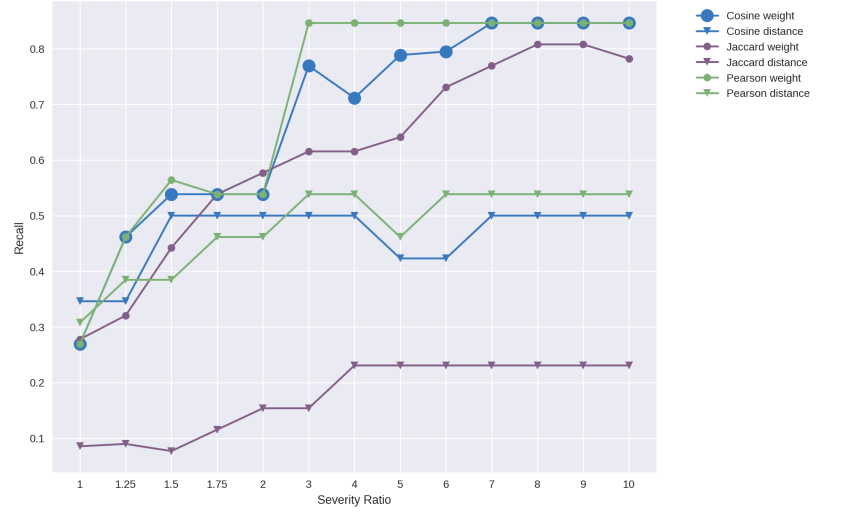
(b) Average recall

Figure 5.22: MAP and average recall of LOF for INFO anomalies data set under different  $k$  (severity ratio = 4), where x-axis is  $k$ , and y-axis is the value of MAP or recall. We use lines with dots to represent the average weight ranking function (Equation 4.19) and lines with triangles to represent the outlier score ranking function (Equation 4.18).

## 5. EXPERIMENT RESULTS

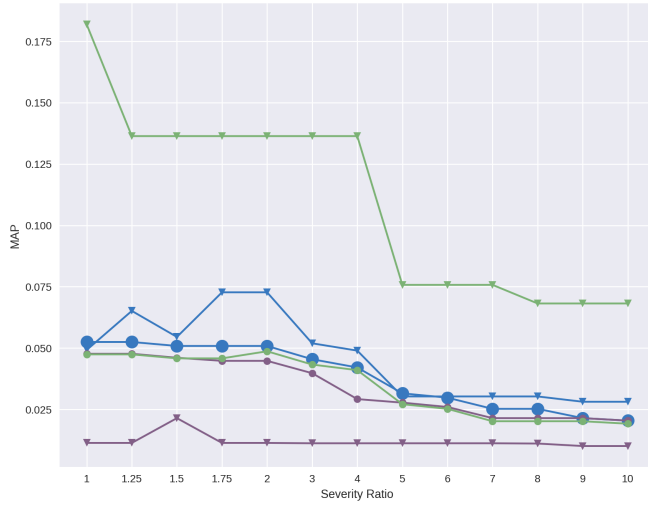


(a) MAP with different severity ratios

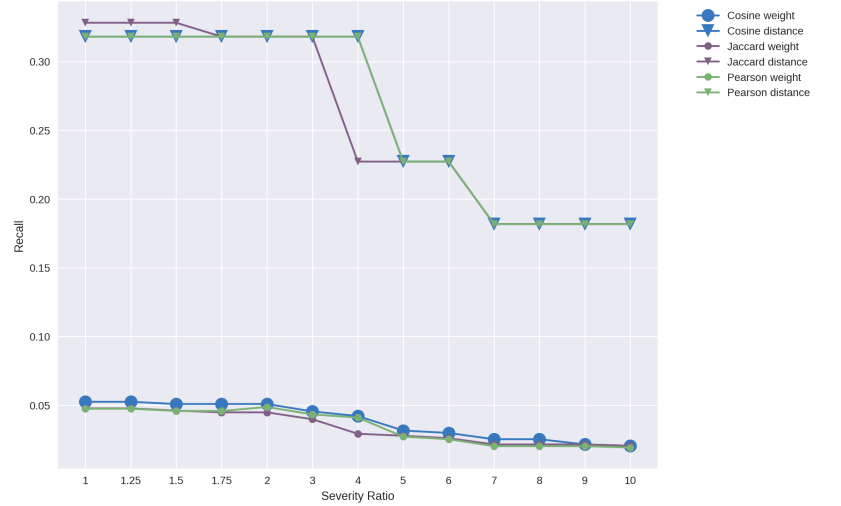


(b) Average recall with different severity ratios

Figure 5.23: MAP and average recall of LOF for original data set using different severity ratios ( $k = 15$ ), where x-axis is the severity ratio, and y-axis is the value of MAP or recall. We use lines with dots to represent the average weight ranking function (Equation 4.19) and lines with triangles to represent the outlier score ranking function (Equation 4.18).



(a) MAP



(b) Average recall

Figure 5.24: MAP and average recall of LOF for INFO Anomalies using different severity ratios ( $k = 15$ ), where x-axis is the severity ratio, and y-axis is the value of MAP or recall. We use lines with dots to represent the average weight ranking function (Equation 4.19) and lines with triangles to represent the outlier score ranking function (Equation 4.18).

Method	Distance metric	Filter threshold	First cluster distance	k	Anomaly cluster distance
Benchmark	Cosine	0.5	0.5	-	0.5
Clustering	Pearson	Equation 5.12	0.4	-	0.4
Nearest neighbor	Pearson	Equation 5.12	-	5	0.4
LOF	Pearson	Equation 5.15	-	15	0.4

Table 5.4: Optimal Parameters for Anomaly Detection Algorithms. The filter threshold is to use to filter out potential anomalies. The first cluster distance is used as the distance threshold by hierarchical clustering to cluster history log data. The anomaly cluster distance is the threshold to cluster potential anomalies in the new log data.

## 5.4 Final Test Result

After choosing the optimal parameters for each anomaly detection shown in Table 5.4, we use test sets (Table 5.2) to evaluate the performance of different anomaly detection algorithms. Like in training set, the INFO anomalies test set is also constructed by changing the anomalous template into INFO severity.

The optimal parameters and the Pearson distance are chosen for each algorithm. Except for benchmark, all other anomaly detection algorithms use a Pearson distance metric. For clustering, the first cluster distance is set to 0.4 and a severity ratio of 4. For nearest neighbor, the  $k$  is chosen to 5 and a severity ratio of 6. For LOF, the  $k$  is set to 15 and a severity ratio of 4.

The final result is shown in 5.5, clustering provides a MAP of 0.359 and a recall of 0.547, but a total recall of 0.952 if the Old Bug Cluster also consider as captured. The reason why benchmark clustering and LOF fail to capture some anomalies in original test set is that anomalies are already seen before the the release in *Release14*, *Release15* and *Release16*. By comparing the distance of multiple neighbors, nearest neighbor distance method can capture recurrent anomalies in these three releases. However, nearest neighbor is not good at capturing INFO anomalies.

Considering the fact that INFO anomalies are quite rare in Adyen releases (only 5 INFO anomalies out of 77 anomalous releases within 5 month), the nearest neighbor distance method is suggested for Adyen data set. The nearest neighbor is capable of providing a more robust performance in anomaly detection even for recurrent anomalies.

## 5. EXPERIMENT RESULTS

---

Ranking function	Method	Original test set			INFO anomalies test set		
		MAP	Recall	Overall Recall	MAP	Recall	Overall Recall
Average weight	Benchmark	0.359	0.548	0.952	<b>0.082</b>	<b>0.333</b>	<b>0.452</b>
	Clustering	0.359	0.548	0.952	0.070	0.321	0.405
	Nearest neighbor	<b>0.520</b>	<b>1.000</b>	<b>1.000</b>	0.060	0.321	0.321
	LOF	0.413	0.548	0.809	0.055	0.321	0.321
Outlier score	Benchmark	0.290	0.333	0.595	0.084	0.333	<b>0.452</b>
	Clustering	0.274	0.333	0.595	0.082	0.321	0.405
	Nearest neighbor	<b>0.307</b>	<b>0.619</b>	<b>0.619</b>	<b>0.088</b>	<b>0.417</b>	0.416
	LOF	0.023	0.071	0.333	0.059	0.143	0.143

Table 5.5: Test Set Results. The ranking function in the first column only indicate the ranking function used by potential anomalous clusters. The average recall is computed using the rankings of **potential anomalous clusters**. The average overall recall is computed using the rankings of both **potential anomalous clusters** and **known clusters**.

## Chapter 6

---

# Experiment Thoughts

In this chapter, we discuss our experiment thoughts with the setting of the pipeline and some ideas beyond experiments.

### 6.1 Data Set

Collecting log data is the most time-consuming process as there are millions of release logs in one host. Moreover, for each anomalous release, two more history releases need to be collected. We collect logs in the monitoring host by querying the *LogSearch* website. The labeling of log events is done manually. Although there are in total 77 anomalous release claimed by developers from September 2017 to February 2018, some anomalies are not related to release or false positives.

#### 6.1.1 Release Instances

In the end, we use 20 releases with their previous two history releases. In total, 60 releases from 17 Live hosts are in as our data set. We call one monitoring release and its previous two releases as one **release instance**. Each release instance consists of logs from the same host. Because each host has different applications and settings of the environment, they generate different numbers or types of logs. At first, we are worried that the model and parameters designed for one host may not be suitable for the rest. This is also the main reason why we want to use anomaly detection algorithms with fewer parameters that can be extended to other hosts.

Each release consists of a different number of logs, from twenty thousand logs to more than 9 million logs. Even though we have many logs in one release instance, we still do not have enough log data to cover all hosts in Adyen platform. We divide our 20 release instances into training and test sets with a ratio of 13:7. Moreover, the training set and test set only have one shared host. This can be used to check if anomaly detection can be extended to unseen hosts.

Because we have limited release instances that are available in the experiments, the training set and test set have different distributions of anomalies. For instance, there are two INFO anomalies in the original training set and zero in the original test set. There are two

recurrent anomalies in the original training set and three in the original test set. We want to make our training set as difficult as possible. Thus, giving The test set itself consists of two recurrent anomalies, and we construct INFO anomalies data set based on the original test set. Therefore, we believe the test set as long with its INFO anomalies version can help us to evaluate the actual performance of the pipeline.

### 6.1.2 Artificial Data Sets

To address RQ3, two artificial data set are constructed. Those two artificial data sets help us to evaluate the performance of different parameters or algorithms more thoroughly.

The INFO anomalies data set are used in parameters tuning section to see if the current value would overlook INFO anomalies. However, most anomalies have high severities. Within 5 months, we only spot 5 INFO anomalies out of 77 anomalous releases of Adyen. Therefore INFO anomalies only account for **6.5%** of the total number of anomalies. This ratio may different in other log data. When we make the compromise between INFO anomalies and high severity anomalies, we should consider this ratio.

Recurrent anomalies are more frequent than INFO anomalies in our data set. Within our data sets, we have 5 release instances out of 20 have a recurrent anomaly. Therefore, **25%** of the release instances might have recurrent anomalies, though this data is not accurate. There are fewer data points in the *new log data* because we postpone the starting time of *new log data*. In other words, we also delay the ending time of *history log data*. This may reduce the number of log event sets in the new log data, making it easy for anomaly detection to capture anomalies. We would suggest using a more sophisticated error injection method to produce recurrent anomalies, without affecting the distribution of the noise.

## 6.2 Extracting Features from Logs

### 6.2.1 Log Abstraction Methods

Although we use IPLoM to abstract raw logs directly, it is also applicable to apply more sophisticated online log abstraction methods in the literature to reduce the processing time of abstraction. In Adyen, other on-going projects are using the source code to generate log templates. In theory, this should be more accurate if the source code is available. If we use the source code to generate log templates beforehand, we can match incoming logs into the templates directly. The drawback of this method is that we need to generate templates again after release.

### 6.2.2 Feature Construction

The logs are unstructured in the Adyen data set. Extracting appropriate features from the logs is essential. The most natural way is to construct feature vectors from time windows during release to capture anomalies. However, this method needs labeling each time window which can be troublesome. Feature vectors constructed using log sequences are quite intuitive for developers. We can map the original log sequences in plain-text directly.



Besides, log sequences represent the process of one task. Log event sets are simplified log sequences, and we use log event sets in anomaly detection. Because we ignore the order of the sequence in our pipeline, we are trying to capture combination anomalies. One of the problems that we meet in this project is that there are always previously unseen log events in new log data. By putting the log events into the sequences, we can calculate the distance of different log event sets.

### 6.2.3 Severity Ratio

Different developers have different opinions of what is WARN and ERROR. The severity ratio around 5 is suggested in Adyen data set. This indicates that high severities log event indeed have a higher chance to be anomalies. This severity ratio should be tuned separately with other log data, by checking experimental results.

### 6.2.4 Logs of Different Applications

Different Java applications in the platform have different significance to the software. A WARN logs of back-office might be less important than those from core application. As the anomalous data is scarce in releases considering the total number of logs, this value might be hard to mine statistically. Different customized weights can be assigned to the log templates by developers themselves.

## 6.3 Anomaly Detection

### 6.3.1 Distance Metrics

In total, we test six different distance metrics in experiments. We exclude three distance metrics in the beginning because they cannot cluster similar log event sets together. Because we want to measure the actual performance of algorithms, all experiments are done using three distance metrics: Cosine, Pearson, and Jaccard. There might also be other suitable distance metrics. Because we need to retrieval most anomalous clusters in the final output, our problem is similar to an information retrieval problem. Besides, we also have a sparse matrix and need to measure the similarity of the sparse vectors. Therefore we only test metrics that are popular in information retrieval.

### 6.3.2 Anomaly Detection Algorithms

We explore the performance of three anomaly detection algorithms, including hierarchical clustering, nearest neighbor distance, and LOF. LOF is not robust enough as shown in Table 5.5 and Figure 5.21a because the performance of LOF is quite sensitive to the value of  $k$ . The  $k$  of LOF might need to be tuned for each host separately.

We use a different filtering threshold (Equation 5.12) from the constant distance value 0.5 that used by benchmark. This method allows us not to neglect the anomalies when it is of recurrent anomalies or INFO anomalies as our filter rule is less strict.

### 6.4 Log Practice

In this section, we discuss the possible improvement that would help with monitoring release logs.

#### 6.4.1 Avoid Similarly Log Statement

Similar log statements in different logging points are possible to make their logs end up with the same template. Similar logging statements may hinder the performance of the anomaly detection or confuse other developers. Besides avoiding adding unnecessary logging points, developers are suggested to use different logging messages in different logging points.

#### 6.4.2 More customized Severity Levels

We suggest that companies use more customized severity levels. For instance, Adyen now uses three customized severity levels: INFO, WARN, and ERROR. WARN indicates a problem is caused by external parties, such as misuse of APIs, or wrong operation of the merchant account. ERROR represents a problem caused by Adyen system, which should be more critical.

It is possible to add more customized severity levels. In this way, we can distinguish a mild WARN from a moderate WARN. Moreover, we can also use more types of severity weights in our pipeline. However, using more severity levels also requires the developers to understand correctly of the meaning of logging statements. Therefore, they can choose the suitable severity level for each logging statement.

## Chapter 7

---

# Conclusions and Future Work

This chapter gives an overview of our contributions to this project. After this overview, we discuss the limitations of this project and draw some conclusions based on the experimental results. Finally, some ideas for future work are discussed.

### 7.1 Contributions

The main contribution of the thesis is to develop a pipeline that can detect anomalies in release logs using enterprises data set from Adyen. Inspired by [34], we are the first to address the problem of release logs monitoring on an enterprise data set with different severity levels.

We design a feature extraction step that does not require the input of source code. We use IPLoM to abstract log templates from raw logs. Later, we apply a reconciling step to help further clustering similar templates. To make long sequences comparable with short sequences, we propose a method to cut long sequences based on the threshold of 99-percentile log length or time duration. To improve the performance of proximity-based anomaly detection, we introduce the severity ratio to assign different severity weights to templates.

Besides hierarchical clustering as used in [34], we use nearest neighbor distance algorithm and LOF as the anomaly detection algorithms. We find the optimal values for different anomaly detection algorithms successfully. We also propose two ranking functions that sort the final output clusters. The average weight ranking (Equation 4.19) performs better in finding anomalies which meet with our assumptions (Section 3.4.2).

Finally, we test the performance of the pipeline on Adyen data set. The average recall of nearest neighbor distance of the test set is higher than its average recall of the training set. We choose **nearest neighbor distance** as our recommended algorithm for Adyen. According to the result of the test set, the nearest neighbor distance outperforms the other two algorithms if recurrent anomalies present in the release. For nearest neighbor distance in the test set, the average recall is 100% for the original data set, and 32.1% for INFO anomalies data set. This result shows that we can capture most anomalies if the anomalies are of higher severity. Besides, we can capture around 1/3 of INFO anomalies.

### 7.2 Limitations

There are some limitations of this thesis, which might make the pipeline fail to detect anomalies in release logs.

#### 7.2.1 Data Set

The first problem of the data set is that we only test on the data set of Adyen. It might not be suitable for other log data sets. This pipeline needs to be tested with other log data set in the future. Secondly, we should have collected more release instances in Adyen.

Because of the limited available release instances as discussed in Section 6.1.1, the training set and test set have different distributions of types of anomalies. This average recall of nearest neighbor distance for training set is 76.9%, which is lower than the average recall for the training set. The lower average recall is because of two recurrent anomalies, release instance 3 and 8, in the training set. To find out the reasons for failure, we list the number of new log events that we do not spot in history log data in Table 7.1. The column “# New Log Event” shows the number of previously unseen log events in the current monitoring release. The column “# New WARN or ERROR” represents the number of previously unseen log events of WARN or ERROR.

These two recurrent anomalies, release instance 3 and 8, cannot be detected by all three anomaly detection algorithms. Release instance 3 cannot be captured because of the high number of new log events, according to Table 7.1. Release instance 8 is exceptional because it does not consist of any new log events. Release instance 8 consists of two consecutive releases. The host failed for the first release and restarted a few minutes later after fixing the issues. In this way, history log data in this release instance also consists of log events that are produced by newly released code.

#### 7.2.2 The Number of New Log Events

When we have many new log events as shown in Table 7.1, the anomaly detection algorithms might be affected by the noise and have a poor performance. In future work, it is possible to study the relation between the number of new log events and the performance of the pipeline.

If there are less than 10 log events in each release, we do not need to put them directly into the sequence. Although we detect anomalies on simplified log sequences level, it is also possible that we can detect anomalous log event directly. We can calculate the overall event weight in Equation 4.7 and sort them based on the overall event weight. Detecting directly anomalous events can be more useful if the number of newly added logging statement in each release is quite low.

#### 7.2.3 Log Abstraction

The parameters of IPLoM and its reconcile step need to be tuned again for different log data. After reconciling, some similar logs are still not clustered. The main problem of log

Release Instance ID	# New Log Event	# New WARN or ERROR
1	103	3
2	28	8
3	424	17
4	39	9
5	39	6
6	24	1
7	43	10
8	0	0
9	91	9
10	224	14
11	46	1
12	38	2
13	85	3
14	53	11
15	44	13
16	44	10
17	17	3
18	69	10
19	150	8
20	14	4

Table 7.1: The number of new log events in release instances. Each row in the table represents one release instance, consisting of a monitoring release and two history releases in one host. The column “# New Log Event” represents the number of previously unseen log events in the current monitoring release. The column “# New WARN or ERROR” represents the number of previously unseen log events of WARN or ERROR in the current monitoring release.

abstraction is the JSON format log messages. Although we already use the key fields for IPLoM, it may need a more sophisticated method to extract the JSON templates.

#### 7.2.4 Anomaly Detection Assumptions

The underlying assumptions in this thesis are that those newly seen log events with a higher severity level and less frequent would be more likely to be anomalous. When the assumptions do not hold in some cases, the pipeline may fail to capture the anomalies. For example, when we have INFO anomalies, it is harder for the pipeline to capture it.

Another problem is that we only consider combination anomalies. When there are performance issues or a sudden change in the number of critical log events, our pipeline cannot detect such anomalies. More features such as the number of occurrences need to be taken into consideration.

### 7.3 Conclusions

After feature extraction and anomaly detection in experiments, we can answer the research questions which are proposed in Chapter 1.

***RQ1: Is it useful to apply the weighted vector model to log event sets in monitoring release logs of Adyen by anomaly detection?***

In the test set, we achieve a recall of 100% in the original data set and a recall of 32.1% INFO anomalies data set using the recommend algorithm: nearest neighbor distance method. The pipeline can successfully detect most anomalies and one-third of the INFO anomalies.

We evaluate the performance of the pipeline with different severity ratios. If we set the severity ratio to 1, it is the same as not use the severity ratio. When we set a higher severity ratio, we would achieve higher values of MAP and average recall in the original data set. The result means that the severity ratio indeed improves the performance of the pipeline when compared with setting severity ratio to 1.

***RQ2: What are the optimal parameters and distance metrics for proximity-based anomaly detection algorithms that are used in release log monitoring?***

The optimal parameters are shown in Table 5.4. To conclude, we use Pearson distance metrics for all three methods: clustering, nearest neighbor, and LOF. For clustering distance, we all use a distance of 0.4. For filtering threshold, clustering and nearest neighbor use Equation 5.12, where the filtering score is calculated as the minimum value of 50-percentile of outlier scores between 0.5. LOF uses Equation 5.15 which is using the 50-percentile of outlier score of new log event sets directly.

***RQ3: Is the anomaly detection pipeline robust in the release when assumptions of the anomaly do not hold?***

To test the robustness, we construct INFO anomalies data set and recurrent old anomalies data to test the performance of anomaly detection algorithms. In the test set, the recommend algorithm nearest neighbor can detect 32.1% of INFO anomalies and 100% recall of the recurrent old anomalies in *Release14*, *Release15* and *Release16*. The clustering algorithm performs better on INFO anomalies and has an overall recall of 0.405%. There are also two recurrent anomalies in the training set. However, our pipeline is not able to capture these recurrent anomalies because there are many new log events in those two monitoring releases.

### 7.4 Future work

#### 7.4.1 Data Sets

As discussed in previous sections, the pipeline should also be tested on other log data to evaluate its performance. On the other hands, the number of collected release instances is quite limited. We only use 20 release instances that belong to 16 hosts. When the infrastructure allows, we should conduct a more thorough study on the release logs of other hosts.

### 7.4.2 Abstraction Methods

IPLoM is a batch processing step. In experiments, this step is the most time-consuming step in the pipeline. To overcome this shortcoming, we can use an online algorithm to increase the efficiency. Moreover, extra attention needs to be taken to JSON format messages in the raw logs as it is quite different than normal string messages written by developers.

This pipeline can also be processed using the events parsed from source code, like the on-going project in Adyen to obtain better accuracy.

### 7.4.3 Considering the Numbers of Occurrence

After choosing the log sequences to extract feature, we again need to choose features between message appearance vectors and message counting vectors. The message appearance vectors are bit vectors only which consists of 1 and 0. While message counting vectors are vectors consists the occurrence number of each template. In this thesis, we choose message appearance vectors to simplify the analysis data. However, it is also possible to use message counting vectors which allow the pipeline to detect anomalies related to the sudden change of occurrence number.

### 7.4.4 Updating Labels of the Logs After Releases

In this pipeline, we do not store the information of future labeling, unlike *Logness*. The labeling information can help improve the performance of anomaly detection if a supervised algorithm is chosen in the future.





---

## Bibliography

- [1] Bram Adams and Shane McIntosh. Modern release engineering in a nutshell—why researchers should care. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 5, pages 78–90. IEEE, 2016.
- [2] Charu C. Aggarwal. *Outlier analysis*. Springer, 2017.
- [3] Charu C Aggarwal, Alexander Hinneburg, and Daniel A Keim. On the surprising behavior of distance metrics in high dimensional space. In *International conference on database theory*, pages 420–434. Springer, 2001.
- [4] JAS Almeida, LMS Barbosa, AACC Pais, and SJ Formosinho. Improving hierarchical cluster analysis: A new method with outlier detection and automatic clustering. *Chemometrics and Intelligent Laboratory Systems*, 87(2):208–217, 2007.
- [5] Joop Aué. An exploratory study on faults in web api integration. Master’s thesis, Delft University of Technology, the Netherlands, 2017.
- [6] RP Jagadeesh Chandra Bose and Wil MP van der Aalst. Discovering signature patterns from event logs. In *Computational Intelligence and Data Mining (CIDM), 2013 IEEE Symposium on*, pages 111–118. IEEE, 2013.
- [7] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. Lof: identifying density-based local outliers. In *ACM sigmod record*, volume 29, pages 93–104. ACM, 2000.
- [8] Peter J Brockwell and Richard A Davis. *Introduction to time series and forecasting*. springer, 2016.
- [9] Stefano Ceri, Alessandro Bozzon, Marco Brambilla, Emanuele Della Valle, Piero Fraternali, and Silvia Quarteroni. *Web information retrieval*. Springer Science & Business Media, 2013.
- [10] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

- [11] Seung-Seok Choi, Sung-Hyuk Cha, and Charles C Tappert. A survey of binary similarity and distance measures. *Journal of Systemics, Cybernetics and Informatics*, 8(1):43–48, 2010.
- [12] Kenneth Church and William Gale. Inverse document frequency (idf): A measure of deviations from poisson. In *Natural language processing using very large corpora*, pages 283–295. Springer, 1999.
- [13] Russ Cox. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby,...). URL: <http://swtch.com/~rsc/regexp/regexp1.html> ( : 29 2010 ).2007, 2007.
- [14] Bharvi Dixit. *Elasticsearch Essentials*. Packt Publishing Ltd, 2016.
- [15] Min Du, Feifei Li, Guineng Zheng, and Vivek 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.
- [16] Abdelmoula El-Hamdouchi and Peter Willett. Comparison of hierarchic agglomerative clustering methods for document retrieval. *The Computer Journal*, 32(3):220–227, 1989.
- [17] Peter Evers. Finding relevant errors in massive payment log data. Master’s thesis, Delft University of Technology, the Netherlands, 2017.
- [18] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang 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.
- [19] Ana Gainaru, Franck Cappello, Stefan Trausan-Matu, and Bill Kramer. Event log mining tool for large scale HPC systems. In *European Conference on Parallel Processing*, pages 52–64. Springer, 2011.
- [20] Saeed Ghanbari, Ali B Hashemi, and Cristiana Amza. Stage-aware anomaly detection through tracking log points. In *Proceedings of the 15th International Middleware Conference*, pages 253–264. ACM, 2014.
- [21] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.
- [22] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. An evaluation study on log parsing and its use in log mining. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*, pages 654–661. IEEE, 2016.
- [23] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael 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.

- 
- [24] Anna Huang. Similarity measures for text document clustering. In *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008)*, Christchurch, New Zealand, pages 49–56, 2008.
- [25] Liang Huang, Xiaodi Ke, Kenny Wong, and Serge Mankovskii. Symptom-based problem determination using log data abstraction. In *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*, pages 313–326. IBM Corp., 2010.
- [26] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010.
- [27] Zhen Ming Jiang, Ahmed E Hassan, Parminder Flora, and Gilbert Hamann. Abstracting execution logs to execution events for enterprise applications (short paper). In *Quality Software, 2008. QSIC'08. The Eighth International Conference on*, pages 181–186. IEEE, 2008.
- [28] Zhen Ming Jiang, Ahmed E Hassan, Gilbert Hamann, and Parminder Flora. An automated approach for abstracting execution logs to execution events. *Journal of Software: Evolution and Process*, 20(4):249–267, 2008.
- [29] Dan Jurafsky and James H Martin. *Speech and language processing*, volume 3. Pearson London, 2014.
- [30] Kamal Kc and Xiaohui Gu. ELT: Efficient log-based troubleshooting system for cloud computing infrastructures. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pages 11–20. IEEE, 2011.
- [31] Tatsuaki Kimura, Akio Watanabe, Tsuyoshi Toyono, and Keisuke Ishibashi. Proactive failure detection learning generation patterns of large-scale network logs. In *Network and Service Management (CNSM), 2015 11th International Conference on*, pages 8–14. IEEE, 2015.
- [32] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.
- [33] Chinghway Lim, Navjot Singh, and Shalini Yajnik. A log mining approach to failure analysis of enterprise telephony systems. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 398–403. IEEE, 2008.
- [34] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei 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.

- [35] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. Mining invariants from console logs for system problem detection. In *USENIX Annual Technical Conference*, 2010.
- [36] Adetokunbo Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. A lightweight algorithm for message type extraction in system application logs. *IEEE Transactions on Knowledge and Data Engineering*, 24(11):1921–1936, 2012.
- [37] Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos 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.
- [38] Leonardo Mariani and Fabrizio Pastore. Automated identification of failure causes in system logs. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 117–126. IEEE, 2008.
- [39] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Tracking behavioral changes in distributed systems using Distalyzer.
- [40] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 26–26. USENIX Association, 2012.
- [41] Nasser M Nasrabadi. Pattern recognition and machine learning. *Journal of electronic imaging*, 16(4):049901, 2007.
- [42] Kishore Papineni. Why inverse document frequency? In *Proceedings of the second meeting of the North American Chapter of the Association for Computational Linguistics on Language technologies*, pages 1–8. Association for Computational Linguistics, 2001.
- [43] Thomas Reidemeister, Miao Jiang, and Paul AS Ward. Mining unstructured log files for recurrent fault diagnosis. In *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, pages 377–384. IEEE, 2011.
- [44] Stephen Robertson. Understanding inverse document frequency: on theoretical arguments for idf. *Journal of documentation*, 60(5):503–520, 2004.
- [45] Felix Salfner and Steffen Tschirpke. Error log processing for accurate failure prediction. In *WASL*, 2008.
- [46] Weiyi Shang, Zhen Ming Jiang, Hadi Hemmati, Bram Adams, Ahmed E Hassan, and Patrick Martin. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 402–411. IEEE Press, 2013.

- 
- [47] Ruben Sipos, Dmitriy Fradkin, Fabian Moerchen, and Zhuang Wang. Log-based predictive maintenance. In *Proceedings of the 20th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1867–1876. ACM, 2014.
  - [48] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972.
  - [49] Bill Stackpole and Patrick Hanrion. *Software deployment, updating, and patching*. CRC Press, 2007.
  - [50] Risto 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.
  - [51] Rick Wieman. What does passive learning bring to adyen. Master’s thesis, Delft University of Technology, the Netherlands, 2017.
  - [52] Rui Xu and Don Wunsch. *Clustering*, volume 10. John Wiley & Sons, 2008.
  - [53] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael 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.
  - [54] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael 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.
  - [55] Wei Xu, Ling Huang, Armando Fox, David A Patterson, and Michael I Jordan. Mining console logs for large-scale system problem detection. *SysML*, 8:4–4, 2008.
  - [56] Kenji Yamanishi and Yuko Maruyama. Dynamic syslog mining for network failure monitoring. In *KDD*, 2005.
  - [57] Ke Zhang, Jianwu Xu, Martin Renqiang Min, Guofei Jiang, Konstantinos Pelechrinis, and Hui Zhang. Automated IT system failure prediction: A deep learning approach. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 1291–1300. IEEE, 2016.
  - [58] Qiankun Zhao and Sourav S Bhowmick. Association rule mining: A survey. *Nanyang Technological University, Singapore*, 2003.