# Logging Statements Analysis and Automation in Software Systems with Data Mining and Machine Learning Techniques

by

Sina Gholamian

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2022

**Examining Committee Membership**

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

Supervisor: Paul Ward
Associate Professor, Dept. of Electrical & Computer Engineering,
University of Waterloo

Internal Member: Mark Crowley
Assistant Professor, Dept. of Electrical & Computer Engineering,
University of Waterloo

Internal Member: Wojciech Golab
Associate Professor, Dept. of Electrical & Computer Engineering,
University of Waterloo

Internal-External Member: Samer Al-Kiswany
Assistant Professor, David R. Cheriton School of Computer Science,
University of Waterloo

External Member: Ding Yuan
Associate Professor, Dept. of Electrical & Computer Engineering,
University of Toronto

## Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

## Abstract

Log files are widely used to record runtime information of software systems, such as the timestamp of an event, the name or ID of the component that generated the log, and parts of the state of a task execution. The rich information of logs enables system developers (and operators) to monitor the runtime behavior of their systems and further track down system problems in development and production settings.

With the ever-increasing scale and complexity of modern computing systems, the volume of logs is rapidly growing. For example, eBay reported that the rate of log generation on their servers is in the order of several petabytes per day in 2018 [17]. Therefore, the traditional way of log analysis that largely relies on manual inspection (*e.g.*, searching for *error/warning* keywords or *grep*) has become an inefficient, a labor intensive, error-prone, and outdated task. The growth of the logs has initiated the emergence of automated tools and approaches for log mining and analysis. In parallel, the embedding of logging statements in the source code is a manual and error-prone task, and developers often might forget to add a logging statement in the software's source code.

To address the logging challenge, many efforts have aimed to automate logging statements in the source code, and in addition, many tools have been proposed to perform large-scale log file analysis by use of machine learning and data mining techniques. However, the current logging process is yet mostly manual, and thus, proper placement and content of logging statements remain as challenges. To overcome these challenges, methods that aim to automate log placement and content prediction, *i.e.*, *'where and what to log'*, are of high interest. In addition, approaches that can automatically mine and extract insight from large-scale logs are also well sought after.

Thus, in this research, we focus on predicting the log statements, and for this purpose, we perform an experimental study on open-source Java projects. We introduce a log-aware code-clone detection method to predict the *location* and *description* of logging statements. Additionally, we incorporate natural language processing (NLP) and deep learning methods to further enhance the performance of the log statements' description prediction.

We also introduce deep learning based approaches for automated analysis of software logs. In particular, we analyze execution logs and extract natural language characteristics of logs to enable the application of natural language models for automated log file analysis. Then, we propose automated tools for analyzing log files and measuring the information gain from logs for different log analysis tasks such as anomaly detection. We then continue our NLP-enabled approach by leveraging the state-of-the-art language models, *i.e.*, *Transformers*, to perform automated *log parsing*.

**Acknowledgements**

This thesis would not have been possible without the valuable advice and the help of several individuals who contributed and extended their valuable assistance in the fulfillment of this thesis. First of all, I would like to thank my supervisor, Professor Paul A. S. Ward, for his guidance and his support throughout my Ph.D. program, which led to the submission of this thesis. His great motivation and accurate view of the research area made a invaluable impression on me and I have learned so much from him.

In addition, thanks to the rest of the colleagues in the Shoshin Research Group at the University of Waterloo for giving me unforgettable help and for sharing with me their helpful and inspiring experiences.

Finally, I would like to express my sincere gratitude to my parents, Ahmad and Forough, and also my sisters, Sara and Samira, for their never-ending love and irreplaceable support throughout my life.

## Dedication

I dedicate this thesis to my parents, Ahmad and Forough, who offered me unconditional love and support throughout my life. It is also dedicated to my siblings, Sara, and Samira, who always supported me throughout my life.

# Table of Contents

# List of Figures

# List of Tables

# List of Abbreviations

ACM: Association for Computing Machinery see p. 18

DL: Deep Learning see p. 135

IEEE: Institute of Electrical and Electronics Engineers see p. 18

LACC: Log-Aware Code Clone detection see p. 96

LACCP: Log-Aware Code Clone detection Plus see p. 121

LM: Language Model see p. 132

LPS: Log Print/Printing Statement see p. 11

LSD: Log Statement Description see p. 120

MD: Method Definition see p. 86

ML: Machine Learning see p. 2

NLP: Natural Language Processing see p. 186

SOTA: State of the Art see p. 5

# Part I
# Prologue

# Chapter 1

# Introduction

Software systems are pervasive and play important and often critical roles in the society and economy such as in airplanes or surgery room patient monitoring systems. Gathering feedback about software systems' states is a nontrivial task and plays a crucial role in systems diagnosis in the case of a failure. In the interest of higher availability and reliability, software systems regularly generate *log files* of their status and runtime information. Developers insert logging statements into the source code which are then printed in the log files, also known as *execution logs* and *event logs* [91]. Then, at a later time, while the system is running or postmortem, developers or operators would analyze the log files for various tasks. For example, the content of log files has been studied to achieve a variety of goals such as anomaly and fault detection [311, 167, 116, 107], online or postmortem performance and failure diagnosis [323, 234, 291, 340, 338], pattern detection [46, 217, 300], profile building [300], business decision making [62], and user's behavior observation [187].

Conventionally, software developers and practitioners apply testing and monitoring techniques to analyze the software systems. System's testing happens during the development phase by developers, while practitioners utilize system monitoring techniques to understand the behavior of the system in the deployed environment [73]. As such, it is a common practice to have running programs report on their internal state and variables, through log files that developers, system administrators, and operators can analyze [65] for different purposes. This continuous cycle of development and deployment of the software and looking at the system logs has also initiated and thrived adjacent fields of research such as DevOps [165, 95, 108]. That is, the importance of log analysis and its computational intensity has also brought in other tools to scale up the effort. For example, considering the advancements of machine learning (ML) and artificial intelligence (AI) and the vast size of the log files, researches have proposed the use of ML for automated operation analysis

(AIOps) [28] of execution logs. From the commercialization perspective, the widespread need for log analysis has also contributed to the emergence of commercial products such as Splunk [43] and Elastic Stack [30]. Splunk makes the large-scale logs accessible by extracting patterns and correlating system metrics to diagnosing problems and provide insight for business decisions. Elastic Stack [30], a.k.a. ELK, consists of three different subsystems of **E**lasticsearch [22], **L**ogstash [21], and **K**ibana [31], works to ingest and process logs from different sources by Logstash, in a searchable format accomplished by Eleasticsearch, and Kibana lets users visualize data with charts and graphs.

From the system's diagnosis perspective, the information provided through the logging statements combined with other system metrics, such as CPU, memory, and I/O utilization, serves an important role in anomaly detection and understanding and diagnosing the system's runtime behavior in the case of a failure. Despite the tremendous potential value hidden in execution logs, the inherent characteristics of logs, such as their heterogeneity and voluminosity [73], make the analysis of them difficult on a large scale and poses several challenges. Some of the associated challenges with logging statements and their analysis in software systems are:

❶ Providing proper logging statements inside the source code remains a manual, *ad-hoc*, and non-trivial task in many cases [191] due to the free-form text format of log statements and lack of a general and well-established guideline for logging.

❷ As the size of computer systems increases and software becomes more complex and distributed, manual inspection of log files becomes cumbersome and impractical, and it calls for automated analysis of logs.

❸ Log data can be heterogeneous and voluminous, as within a large software system multiple subsystems may potentially generate a plethora of logs in different formats.

❹ Developers and automatic logging tools, that aim to automate the addition or enhancement of logging statements in the source code, always face challenging questions of *"what, where, how, and whether to log?"*.

Considering the aforementioned challenges, prior and ongoing research has made numerous efforts to mine and understand log statements in the source code and execution log files to either gain more insights about logging practices, troubleshoot the software, or automate the logging process [325, 78, 346]. Figure 1.1 depicts a framework in which the creation process and analysis of log files are illustrated. After the system's architecture is decided and programmers implement the source code with logging statements, the operators run the system with selecting proper runtime configuration parameters (*e.g.*, logging

verbosity level in Log4j [15]). While the software is running, events that are logged in the source code generate records within the log files. Next, administrators (, practitioners), and automated log analyzer tools may review the files and feedback the outcome to the designers, programmers, and operators to make adjustments to the architecture, source code, and system configuration if needed, respectively.

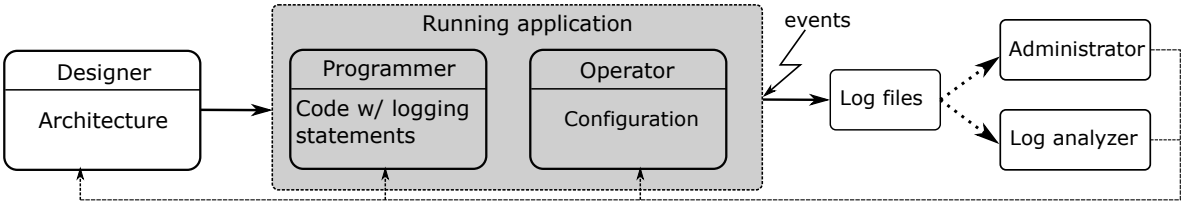



Figure 1.1: Framework for creation and analysis of log files [273].

## 1.1 Research Motivation

This thesis aims to gain more insight into the current software systems logging practices and propose automated approaches for logging and information gain from execution logs. We observe that although logging statements are generally considered to be in unstructured text format, they still bear some structure (*i.e.*, semi-structured, such as timestamp, verbosity, *etc.*), which can be leveraged for automated log analysis. Specifically, this thesis pursues the following research hypothesis:

**Research Hypothesis**

Current developers' logging practices can be learned and leveraged in the development of automated log prediction tools. Additionally, the semi-structured format of logs and the information gained from execution logs can be utilized to develop tools for automated log analysis to support developers' and practitioners' efforts in software testing and monitoring.

## 1.2 Thesis Outline

In the following, we provide an outline for each chapter of this thesis.

### 1.2.1 Chapter 2: Literature Review

In this chapter we survey the state-of-the-art (SOTA) logging research in software systems and its applications. We categorize the prior work into two main categories:

- **Logging source code:** discusses the research that aims to understand log statements and uncover patterns and issues for improving and automating the logs in the source code.
- **Log files:** explains the research that aims to gain information from the logs to improve automated log analysis for various applications, *e.g.*, log file anomaly detection.

Our findings from the literature review enable us to observe the missing pieces of the puzzle for improvement and future work in the logging research for software systems.

### 1.2.2 Chapter 3: Research Plan for Log Prediction

In this chapter, we review our research plan and steps for the application of similar code snippets, *i.e.*, code clones, and natural language processing for log statement location and content automation.

### 1.2.3 Chapter 4: Clone Detection Background

In this chapter, we provide the necessary background on source code clones. Clone detection is the procedure of locating exact or semantically similar pieces of source code within or between software systems which we apply in the following chapters for log-aware clone detection.

### 1.2.4 Chapter 5: Log Location Prediction

In this chapter, we explain our approach for predicting log statement locations based on the source code clones. We perform an experimental study on open-source Java projects and show that code clones follow similar logging patterns. We then apply this feature to propose a log-aware code-clone detection method for log statement prediction.

### 1.2.5 Chapter 6: Log Content Prediction

In this chapter, we improve on the performance of log-aware clone detection to predict the *location* and *description* of logging statements. Additionally, we incorporate natural language processing (NLP) deep learning methods to further enhance the performance of the log statements' description prediction. We evaluate the performance of the predictions with BLEU and ROUGE scores.

### 1.2.6 Chapter 7: Cost and Gain from Logs

To evaluate the performance overhead and storage cost of logging, in this chapter, we present the result of our experimental study on seven Spark benchmarks to illustrate the impact of different logging verbosity levels on the execution time and storage cost of distributed software systems. We also evaluate the log effectiveness and the information gain values, and study the changes in performance and the generated logs for each benchmark with various types of distributed system failures. Our research draws insightful findings for developers and practitioners on how to set up and utilize their distributed systems to benefit from the execution logs.

### 1.2.7 Chapter 8: Naturalness of Logs

We discuss natural language attributes of log files in this chapter and explain how these attributes can be utilized for automated log analysis. We begin with the hypothesis that log files are natural and local and these attributes can be applied for automating log analysis tasks. We guide our research with six research questions with regards to the naturalness and localness of the log files, and present a case study on anomaly detection and introduce a tool for anomaly detection, called *ANALOG*, to demonstrate how our new findings facilitate the automated analysis of logs.

### 1.2.8 Chapter 9: Natural Language Models for Log Parsing

Previously, we showed that NLP characteristics can be applied for automated analysis of log files as log records can be looked at as natural language sequences. Thus, in this chapter, we leverage the latest advances in NLP models and employ "Bidirectional Encoder Representations from Transformers (BERT)" NLP models to enable more accurate automated log file parsing.

### 1.2.9 Chapter 10: Conclusions and Future Work

We conclude and summarize our research in this chapter and give directions for future avenues of work.

### 1.2.10 Chapter 11: Summary of Publications

Chapter 11 provides a summary of publications from this thesis.

## 1.3 Contributions

This thesis proposes novel approaches for log location and content prediction. In addition, we provide insight on automated analysis of logs which provides developers and practitioners with more effective troubleshooting. We summarize our contributors as follows:

- In Chapter 2, we provide a comprehensive and systematic literature review of current logging practice, challenges associated with logging, and automated logging and log mining.
- In Chapters 3, 4, 5, and 6 we initially explain the background for source code clone detection and then provide a novel approach for automating the log statements and their content.
- In Chapter 8, we investigate the naturalness and locallness of execution logs and demonstrate how these attributes can be leveraged for automated log analysis.
- We study costs and benefits associated with logging in distributed systems in Chapter 7, and provide insight on the information gain from logs in the case of failures.
- Finally, in Chapter 9, we provide novel BERT-based log parsing approach, which utilizes the natural language characteristics of the logs.

## 1.4 Closing Remarks

We discussed the importance of logs, how they help developers and practitioners, and the existing challenges to make logs more effective. As we conclude this chapter, we continue with the survey of the prior work in the next chapter.

# Part II
# Systematic Literature Review and Mapping

# Chapter 2

# A Comprehensive Survey of Logging in Software: From Logging Statements Automation to Log Mining and Analysis

***Abstract-*** Logs are widely used to record runtime information of software systems, such as the timestamp and the importance of an event, the unique ID of the source of the log, and a part of the state of a task's execution. The rich information of logs enables system developers (and operators) to monitor the runtime behaviors of their systems and further track down system problems and performs analysis on log data in production settings. However, the prior research on utilizing logs is scattered and that limits the ability of new researchers in this field to quickly get to the speed and hampers currently active researchers to advance this field further. Therefore, this chapter surveys and provides a systematic literature review of the contemporary logging practices and log statements' mining and monitoring techniques and their applications such as in system failure detection and diagnosis. We study a large number of conference and journal papers that appeared on top-level peer-reviewed venues. Additionally, we draw high-level trends of ongoing research and categorize publications into subdivisions. In the end, and based on our holistic observations during this survey, we provide a set of challenges and opportunities that will lead the researchers in academia and industry in moving the field forward.

**Keywords:**
survey, systematic literature review (SLR), software systems, logging, log statement, log

file, log automation, log analysis, log mining, logging cost, anomaly detection, failure detection and diagnosis

**A reversion of this chapter is under review at the IEEE Journal of Transaction on Software Engineering [125].**

## 2.1 Introduction

As the size of computer systems increases, the manual process of developers placing logging statements into the source code and administrators reviewing the log files and detect problems negatively affecting the system becomes less practical and less effective. Consequently, being able to automatically detect logging points and insert appropriate logging statements in the source code, as well as systematically detecting system issues are very beneficial and high-in-demand research topics [346, 339]. Thus, researchers have dedicated a significant amount of studies in the area of software logging and log analysis techniques throughout the last decade [73] to propose various approaches, including automated analysis and application of machine learning to process large-scale log files. However, after reviewing the prior work, we noticed what has not been addressed is a clear review of the current progress in software systems' logging and log analysis research. It is, therefore, hard for researchers to recognize how their current and future work will fit in the big picture of present-day logging research. Understanding where we are at the moment and creating a snapshot of the current research is a fundamental step towards understanding where we should go from here and what the necessary next steps of the research would be. Learning from our own experiences and the obstacles that we have to go through to holistically picturize the current research in the field, this survey aims to pave the road obstacles and provide a methodological review of logging, its practices, and its automation techniques and tools for software systems. Additionally, in this survey, we review the current state of logging in software to discover solutions for the aforementioned challenges and highlight the next steps for future research efforts. We review and study a vast number of peer-reviewed conference and journal papers from related research areas including software and distributed systems, dependability, and machine learning. Moreover, we aim to build knowledge [63] and trends by connecting and combining findings from multiple research. Thus, we examine and categorize the prior research for *logging costs*, *logging practices*, *automation of log analysis*, and *efforts to automate the insertion and improvement of logging statements inside the source code*. Finally, we provide *trends and opportunities for future work* based on the insights we gain during this survey. For each research, we provide its *1) aim*, *i.e.*, the problem it is trying to address, *2) experimentation*, *3) results and findings*,

*4) advantages*, *i.e., pros*, and *5) disadvantages*, *i.e., cons*. Before we dive deeper into the survey, we will review a few important vocabulary in the following.

### 2.1.1 Terminology

**Log printing statement (LPS)**. LPSs are the log statements in the source code added by developers. We use *"log printing statement"*, *"log statement"*, and *"logging statement"* interchangeably, as the prior work has used all of the variations [116, 339, 117, 123, 192].

**Log message.** A log message, typically a single line, is the output of the LPS in the log file. Prior work also makes a subtle distinction between a log message and a **log entry**, and defines a *"log entry"* or a *"log record"* as a line in the log file composed of a log header and a log message [110]. Log header contains timestamp, verbosity level, and source component, and its format is usually defined by the logging framework, *e.g.*, *log appenders* [37], whereas a log message is written by the developer and consists of the amalgamation of the static part of the log message from the source code and the dynamic value of the variables during the runtime. For our purpose in this survey, log message, log entry, log record, and log event are used interchangeably [250, 192, 115, 183].

**Log file.** Log file(s) is a collection of log messages stored on a storage medium, also called *"event logs"*, and *"execution logs"*, or simply just *"logs"* [115, 183, 116]. In most cases, these terms can be used interchangeably, and we commonly use log file(s) as an umbrella term to cover the different naming variations throughout the survey. As a minor point, in special cases that we mean to refer to a set of log lines in general (*i.e.*, without binding them to specific files), it is more appropriate to refer to them as execution logs or log records (*e.g.*, *execution logs* are used for anomaly detection).

Additionally, we might refer to computer (computing) systems and software systems interchangeably on some occasions throughout this survey with regards to logs, meaning that the log messages in the log files are generated from log printing statements within the source code of the software systems (for various software or hardware related events, concerns, or issues), which are also an artifact of computer systems as an umbrella for software, hardware, and anything in between.

### 2.1.2 Research Questions

While we review the prior literature, we aim to purse and answer the following research questions:

- **RQ1:** How to systematically review and categorize prior logging research into different topics?
- **RQ2:** What are the publication trends based on venues, topics, and years?
- **RQ3:** How the research in each topic can be systematically compared with their *approaches*, *pros*, and *cons*?
- **RQ4:** What open problems and future directions are foreseeable for logging research?

With the pursual of the aforementioned RQs, we ensure to follow the established evidence-based software engineering (EBSE) [172, 177] paradigm for our literature review. As a contributing improvement, our survey combines and benefits from the advantages of both systematic literature review (SLR) and systematic mapping (SM) paradigms. Prior research indicates the main differences between SM and SLR are that SM methodology is broader and more based on qualitative measures, while SLR focuses on narrower research questions and quantitative measures [237]. While being comprehensive, *i.e.*, SM, our survey also provides details on the experimentation and results of each primary study, *i.e.*, SLR. In summary, RQ1, RQ3, and RQ4 qualitatively assess the prior literature into different research categories base on different topics, *i.e.*, systematic mapping (SM) [309], whereas RQ2 quantitatively measures the publications based on venues, topics, and years, and RQ3 provides details aligned with SLR data extraction methods for each study, *e.g.*, *the aim*, *experiments*, *results*, and *findings* [237].

### 2.1.3 Survey Organization

The rest of this survey is organized as follows. Section 2.2, prior to answering the RQs, we provide the background for log statements, messages, and files. In Section 2.3, we provide our findings for RQ1 and categorize the prior logging research. In Section 2.4, we present our findings for RQ2 and present the publication trends for different topics, years, and venues. Then, Section 2.5 reviews the prior research in each category of logging in details and provide our findings for RQ3. In Section 2.6, we provide our findings for RQ4 and describe open problems and opportunities for future work, and Section 2.7 concludes the survey.

## 2.2 Log statements and Log Files

Logging is the process of recording and keeping track of the *events of interest*, *e.g.*, to developers, practitioners, system admins, and end users, while the software is running. As such, log messages aim to achieve this goal and record the events of interest that happen during the software system's execution and store them in the log files. Generally, the logging process starts with software developers (*i.e.*, programmers) include logging statements with description, variables, and verbosity levels (Figures 2.1 and 2.2) into the source code. Then, while the software is running, the logging statements are ***logged***, if appropriate configurations (*e.g.*, verbosity level) are enabled. In the simplest case, log messages are written to a single log file. However, in a distributed system, there can be multiple log files in different formats. The focus of our survey is on this type of logs which are also called execution logs or event logs [299]. Event logs are the outcome of logging statements that software developers insert into the source code. Event logging and log files are playing an increasingly important role in computer systems and network management [299, 311, 73], which we will review later in this survey.

### 2.2.1 Transaction Logs

It is also worth mentioning briefly the difference between the execution logs and transaction logs. A transaction log (also called journal) is a record file of the transactions between a system and the users of that system, or a data collection method that naturally captures the type, content, or time of transactions made by a user from interacting with the system. In a database server, a transaction log is a file in which the server stores a record of all the transactions performed on the database [98]. The transaction log is an important component of database servers and cryptocurrency protocols (*e.g.*, blockchain [64]) when it comes to recovery. If there is a system failure, transaction logs are used to revert the database back to a consistent state. In summary, transaction logs act as a ledger to accurately record the transactions in the system which are agreed, shared, and synchronized among all the parties involved, whereas execution logs capture events of interests with different verbosity and severity levels, *e.g.*, DEBUG, INFO, ERROR, *etc.*, which do not necessarily require sharing, agreement (*i.e.*, consensus), or synchronization between different software modules.

### 2.2.2 Log Example

Software developers utilize logging statements inside the source code to gain insight into the internal state of applications amid their execution. In the simplest form, logging statements are *print* statements utilized in different programming languages. In this case, the logging statement may contain a textual part indicating the context of the log, *i.e.*, the *description* of the log, a *variable* part providing contextual information about the event. Figure 2.1 shows an example of logging statements in *C* programming language.

```
printf("Cannot find BPService for bpid=%d", id);
       |        description                | variable
```

Figure 2.1: Log example in *C*.

Logging statements may utilize logging libraries to improve the organization of the logged information. For example, in Java, libraries such as Log4j [15] and SLF4J [29] provide a higher degree of flexibility to the developers. Logging libraries, also called logging utilities (LU) [81] or logging libraries and utilities (LLUs), provide extra features, such as *log level*, which indicate the verbosity and the severity of the logging statement. Log levels help to better distinguish the importance of runtime events and control the number of logs collected on the storage device [229]. For example, less verbose levels, *i.e.*, *fatal*, *error*, and *warn*, are leveraged to alarm the user when a potential problem happens in the system, and more verbose levels such as *info*, *debug*, and *trace* are utilized to record more general system events and information or detailed debugging. In practice, *info* and more verbose levels are utilized during the software development phase by programmers, and *info* or less verbose levels are, by default, for the software deployment phase, as the end user observes. In case more insight about the internal state is needed, end users might enable more verbose logging. An example of a logging statement with library usage for *warn* verbosity level is shown in Figure 2.2.

```
log.warn("Cannot find BPService for bpid=" + id);
  level |          description          | variable
```

Figure 2.2: Log example with Log4j library.

Logging statements are generally saved in log files. Figure 2.3 shows ten lines of logs from Apache Spark [16] execution logs collected in our execution of *k-means* clustering

algorithm [216] on a standalone cluster. In real-world cases, a computing cloud system can generate millions of such log messages per minute [347]. For example, for an online store with millions of customers worldwide, it is common to generate tens of terabytes of logs in a single day [211, 208].

| Log file example | |
|---|---|
| 1 | 20/02/21 13:47:47 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable |
| 2 | 20/02/21 13:47:47 INFO SparkContext: Running Spark version 2.4.4 |
| 3 | 20/02/21 13:47:48 INFO SparkContext: Submitted application: JavaKMeansExample |
| 4 | 20/02/21 13:47:48 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(user); groups with view permissions: Set(); users with modify permissions: Set(user); groups with modify permissions: Set() |
| 5 | 20/02/21 13:47:49 INFO StandaloneSchedulerBackend: Connected to Spark cluster with app ID app-20200221134749-0004 |
| 6 | 20/02/21 13:47:49 INFO StandaloneAppClient$ClientEndpoint: Executor added: app-20200221134749-0004/0 on worker-20200220231425-192.168.210.13-34881 (192.168.210.13:34881) with 8 core(s) |
| 7 | 20/02/21 13:47:50 INFO EventLoggingListener: Logging events to file:/tmp/spark-events/app-20200221134749-0004 |
| 8 | 20/02/21 13:47:51 INFO DAGScheduler: Submitting 933 missing tasks from ResultStage 0 (MapPartitionsRDD[5] at map at KMeans.scala:248) (first 15 tasks are for partitions Vector(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14)) |
| 9 | 20/02/21 13:47:53 INFO CoarseGrainedSchedulerBackend$DriverEndpoint: Registered executor NettyRpcEndpointRef(spark-client://Executor) (192.168.210.12:39482) with ID 3 |
| 10 | 20/02/21 14:03:58 INFO DAGScheduler: Job 12 finished: sum at KMeansModel.scala:105, took 13.099825 s |

Figure 2.3: A log file example from Apache Spark.

As observed in Figure 2.3, a major part of logging messages is unstructured text. Thus, in order to make log files useful and avoid the hassle of manually processing a plethora of log files, the first and foremost step of log processing is the automatic *"parsing"* of log messages, which transforms unstructured logs into structured events. Not to be confused with syntactic parsers in programming languages, which parses source code and confirm whether it follows the rules of the formal grammar, log parser, on the other hand, transforms unstructured raw log files into a sequence of structured events, to enable automated analysis of logs. We review the log parsing process later in Section 2.5.4.

### 2.2.3 Program Traces vs. Logs

The term log is often used to represent the way a program is used (such as security logs), while tracing (not to be confused with *"trace"* log level in logging libraries such as Log4j) is used to capture the temporal sequence of events during a particular execution of a program [228], in contrast to logs which are generally the consolidation of continuous execution of software systems. Tracing is typically performed by an external program/tool that instruments the runtime environment, such as network traffic traces, whereas logs are the direct output of logging statements' execution inside the software. Moreover, while traces are typically structured data, logs are free-form and unstructured text. For example, a trace can contain the software execution paths, the events triggered during the execution, and the value of variables, which are used for debugging and program understanding. Stack traces are common examples that are used for function call tree tracing during development and postmortem debugging. Logs and traces are also different in their analysis difficulty level. Because traces are inserted by external tools, they are straightforward to comprehend during analysis as the trace patterns are set by the same tools that inserted them and will analyze them. In contrast, because log statements are inserted by developers who are typically not the ones who build log analysis tools, thus, log analysis tools face a significant challenge in parsing, and further understanding the semantics of the logs. Some of the well-known tracing systems include Google's Dapper [286], X-Trace [113], Microsoft's Magpie[61], the *black-box* approach [45], and Casper [310].

## 2.3 RQ1: How the prior logging research can be categorized to different topics?

In this section, we provide our findings for the first research question by reviewing the available literature and categorizing the logging research into its topics (and subtopics), which enables us to explain the survey scope that follows next.

### 2.3.1 Survey Methodology

Studying log mining and logging analysis techniques in software systems is a challenging and widespread topic, and there have been numerous prior studies that focus on log analysis. Table 2.1 summarizes the databases and keywords used in our survey, and Figure 2.4 provides the flowchart of the steps in the database selection and reference analysis. We

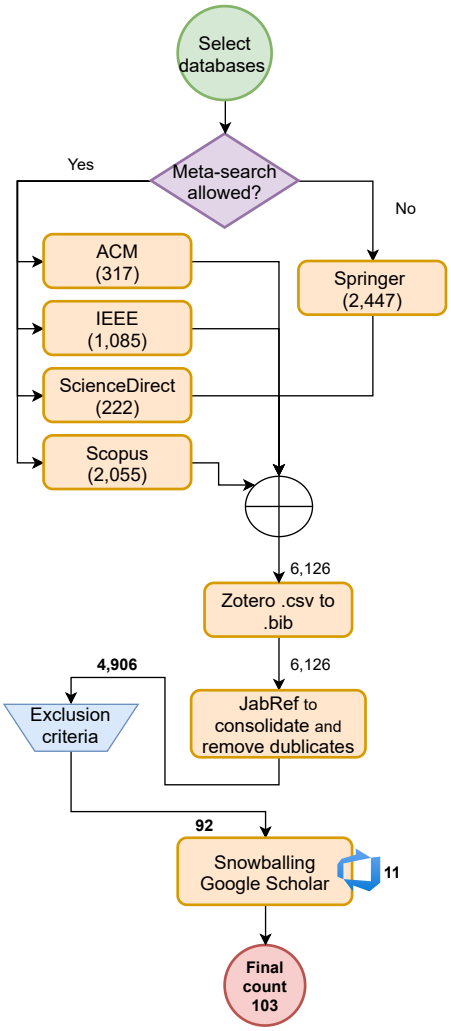



Figure 2.4: This figure provides the steps involved in our survey methodology.

start with the established research databases listed in the table and search for the list of log-related keywords. If the database allows for metadata search (*i.e.*, title, abstract, and keywords), we limit our search to metadata, to avoid the inclusion of numerous unrelated research that has the keywords in their main text. However, for Springer, we could not use meta-search, and thus, the reason behind the high number of returned publications. We use reference management software, such as Zotero [1] and JabRef [2] to facilitate and automate our process of reference consolidation from different sources and duplicate removal.

| | |
|---|---|
| **Research databases** | ACM Digital Library, IEEE Xplore, ScienceDirect (Elsevier), Scopus, SpringerLink, and Google Scholar (snowballing). |
| **Keywords** | 'software AND log AND (statement OR file OR record OR event)' AND 'Publication Date: (01/01/2010 TO 05/31/2021)' |

Table 2.1: Literature review databases and keywords. We searched in various online research databases for different combinations of log-related keywords.

After duplicate removal, our process resulted in 4,906 papers.

For **exclusion criteria**, our focus has been on the full (*i.e.*, not short), recent papers in English in the last decade in the established venues, ranging from very good to flagship ones [3]. After the keyword search, we manually investigated the results and included relevant publications that directly tackle issues associated with log statements or log files, and excluded the ones which had a weak association with logs. Also, if a research project has multiple variations, *i.e.*, a conference paper followed by a more comprehensive journal paper, we only include the more comprehensive version. Additionally, once we find an influential (*i.e.*, highly-cited) paper, we also check all of its references and its citations, *i.e.*, *snowballing search* [308] with Google Scholar. Followed by exclusion criteria, we are narrowed down to 92 publications, and at last, 11 additional references are added with snowballing, bringing the total count to **103** publications.

We categorize the selected publications (103) into **twelve categorizes** after carefully studying them. Our methodology for categorizing the publications has been based on a top-down approach. Meaning that, we first were able to draw categorizes that either focus on logging statements or the ones that focus on log files. We then further narrowed down each category based on its primary focus. Next, we also extracted subtopics for each paper, as usually, publications also partially cover some other related topics in their research. For example, Zhao *et al.* work [339] primarily focuses on log statement automation, but it also covers topics related to log cost analysis. The details for each category is available in Table 2.19 in Section 2.8.

[1] https://www.zotero.org/
[2] https://www.jabref.org/
[3] https://www.core.edu.au/conference-portal

### 2.3.2 Survey Scope

Based on our methodology, the scope of our survey is developed as follows. To the best of our knowledge, there is no prior work that provides a systematic and comprehensive coverage on log mining and automation techniques in software systems, covering different aspects of logging such as mining source code and log files, automating log printing statements in the source code, and their evaluation techniques. There are a few existing surveys on the application of execution logs for anomaly and problem detection [272], system monitoring [73], and instrumentation [77]. We cover the following sections in our survey:

- **Logs and log files.** We explain what are log files, log statements, and log messages, and what sort of applications and analyses they are leveraged for.
- **Logging cost.** We point out the quantitative and qualitative costs and benefits associate with logging.
- **Logging statement mining and automating.** Logging research aims to understand current logging practices and use the findings to improve the log printing statements with automatic log insertion and learning to log techniques. Thus, we review:
  - **Logging code practices.** This section includes studies that empirically or automatically investigate how developers insert logging statements into the software's source code and how the logging evolution and improvement can benefit the usage of the logging code.
  - **Automatic log insertion and learning.** We cover the studies that leverage static code analysis, heuristics, and machine learning techniques to automatically add (or improve) logging statements in the source code to make them more effective in failure diagnosis.
  - **Evaluation.** As we study automatic ways of addition and learning of log statements, we introduce several metrics to measure the effectiveness of the proposed approaches.
- **Mining logs.** This part provides insight on methods and tools to analyze log messages and log files. This can be further divided to log management, log parsing, and their applications.

  - **Log management and maintenance.** Management and collection of logs are important as a pre- or post-step of log analysis.
  - **Log parsing.** To enable log message analysis, we first require to parse the log messages and extract their templates.
  - **Application of logs.** We review a wide range of applications that leverage automated log analysis for various software engineering tasks, such as anomaly detection and failure diagnosis.

- **Emerging applications of logs.** We review the recent special interest in applications of logs in other domains such as mobile devices and big data.

- **Opportunities for future work.** Based on the current log-related research, we comment on the future directions and opportunities for each category of logging research.

As our aim is to provide a comprehensive and end-to-end survey of the logging in software, we have covered the topics of ***mining and automation of logging statements*** and ***mining of log files*** side-by-side in this survey. We have seen that these topics go hand-in-hand and the synergy between them has resulted in more effective logs and logging practices. In fact, the ultimate goal of a mountain of studies for logging statement automation is to ***improve various log mining tasks*** (Figure 1.1). For example, *ErrLog* [324] and *LogEnhancer* [326] automatically introduce new logging statements or add additional variables to the logging statements (*i.e.*, **automation of the logging statements**) to improve the quality of logs, and, subsequently, ***improve log mining tasks*** such as error detection and program diagnosability. In another example, authors of *Log20* [339], an automated log placement tool, explain that the main objective of their log placement tool is to *"disambiguate"* the execution paths, and consequently, improve the effectiveness of log mining methods in *"debugging real-world failures"*. Furthermore, we observe that ignoring the cross-cutting concerns of logging has resulted in log-related issues in the past, such as stale and confusing logging statements, and has hindered effective log file mining [277, 170].

It is also important to mention that there exist additional industry products, with an aggregate market cap beyond $125B, that perform log analysis for various goals such as performance evaluation, cloud monitoring, and data analytics, to name a few: Datadog [32] ($49.7B)[4], Splunk [43] ($27.3B), Elasticsearch [22] ($15.5B), Loggly [35] ($5.7B), Dynatrace [34] ($22.1B), New Relics Inc. [39] ($4.9B), and XpoLog [44]. However, this chapter

[4]Values are collected at the time of this survey from Google Finance. For example for Datadog: https://www.google.com/finance/quote/DDOG:NASDAQ

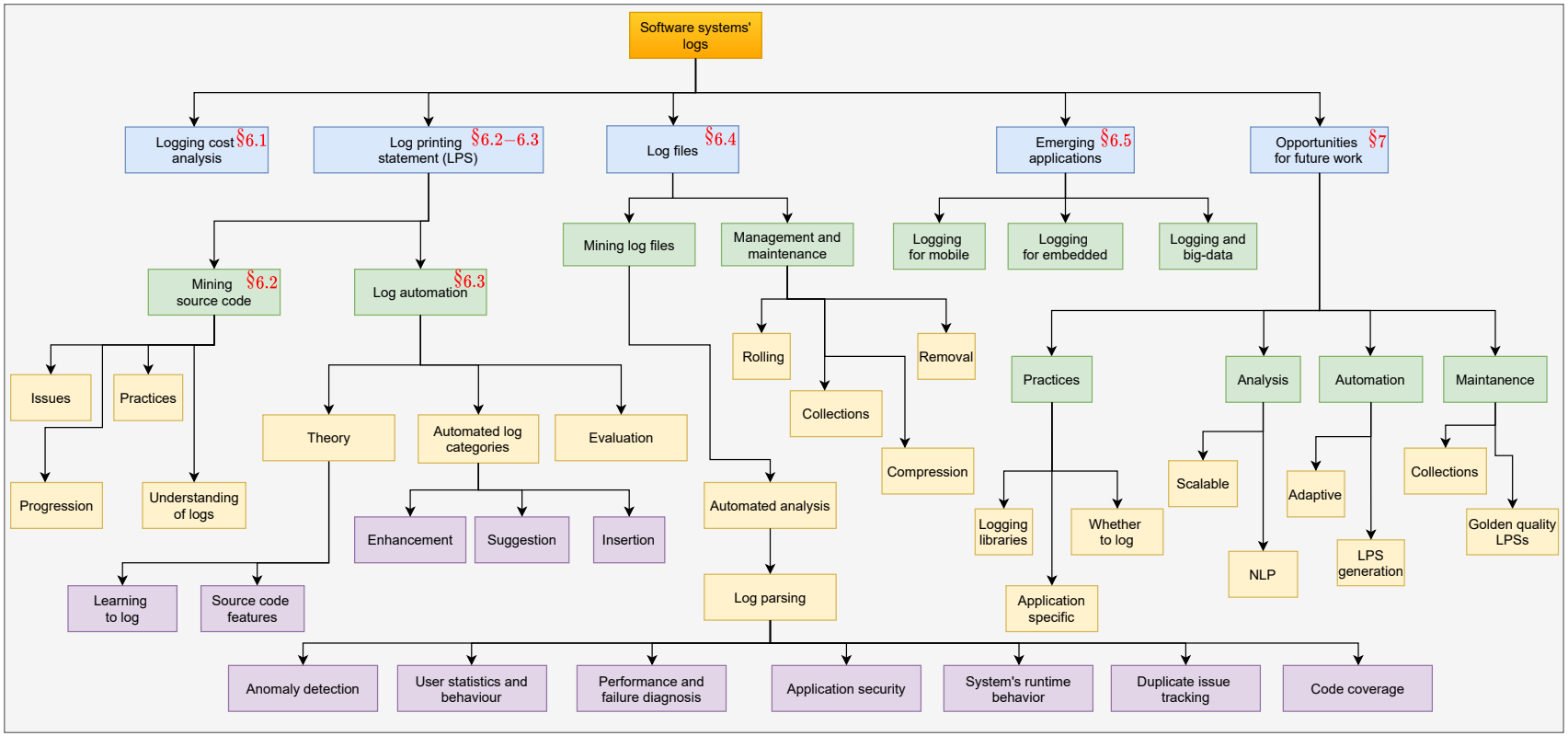



Figure 2.5: This figure provides a taxonomy of the present-day logging research. After providing an introduction on logs and log messages in §2.2, we explain logging costs and benefits in §2.5.1. Log printing statement research covers two sections, §2.5.2-2.5.3, followed by log files' research in §2.5.4. §2.5.5 and §2.6 provide emerging applications and opportunities for future work. We conclude our survey in §2.7.

| No. | Type | Name (# of publications) | Abbr. | References | No | Type | Name (# of publications) | Abbr. | References |
|---|---|---|---|---|---|---|---|---|---|
| 1. | J | Empirical Software Engineering (12) | EMSE | [269, 318, 79, 279, 170, 196, 142, 194, 193, 319, 86, 330] | 10. | C | IEEE International Conference on Data Mining (3) | ICDM | [205, 116, 106] |
| 2. | C | International Conference on Software Engineering (11) | ICSE | [291, 203, 315, 325, 78, 81, 117, 250, 346, 278, 62] | 11. | C | Mining Software Repositories (3) | MSR | [169, 72, 126] |
| 3. | J | IEEE Transactions on Software Engineering (5) | TSE | [202, 210, 192, 96, 209] | 12. | C | International Symposium on Software Reliability Engineering (3) | ISSRE | [65, 84, 251] |
| 4. | J | Journal of Systems and Software (5) | JSS | [60, 112, 222, 115, 277] | 13. | C | ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (2) | ESEC/ FSE | [149, 337] |
| 5. | C | International Conference on Automated Software Engineering (4) | ASE | [200, 82, 208, 145] | 14. | C | International Conference on Software Analysis, Evolution and Reengineering (2) | SANER | [170, 166] |
| 6. | C | USENIX Symposium on Operating Systems Design and Implementation (4) | OSDI | [341, 340, 322, 324] | 15. | C | International Conference on Architectural Support for Programming Languages and Operating Systems (2) | ASPLOS | [323, 321] |
| 7. | C | IEEE/IFIP Conference on Dependable Systems and Networks (4) | DSN | [314, 242, 244, 90] | 16. | C | USENIX Annual Technical Conference (2) | ATC | [212, 104] |
| 8. | C | International Symposium on Reliable Distributed Systems (4) | SRDS | [118, 332, 135, 87] | 17. | C | ACM Symposium on Applied Computing (2) | ACM SAC | [123, 221] |
| 9. | C | ACM Symposium on Operating Systems Principles (3) | SOSP | [339, 311, 338] | | | | | |

Table 2.2: List of related venues for our survey sorted from the most to least number of references. J stands for journal, and C stands for conference, symposium, or workshop. The list of additional venues with only one publication include: arXiv [285, 151], CSUR [77], TOCS [326], TPDS [102], IEEE Software [228], TSMCA [199], TNSM [160], SP&E [175], JCST [349], HiPC [88], CSRD[292], IJCAI [224], NSDI [234], CIKM [294], CCS [107], SIGKDD [198], IMC [258], MASCOTS [57], WSE [296], IWQoS [343], ICWS[148], ICPC [225], ICSEM [54], ICPE [317], ICC [329], Middleware [313], VLDB [187], CNSM [300], APSYS [333], HICSS [274], and COMPSAC [184]. The total number of listed papers is **103**.

mainly focuses on surveying academic works, or peer-reviewed publications from industry. We acknowledge that there is a mountain of work in industry that perform log analysis, however, it is mainly outside the scope of this survey. Additionally, logging is also used in other computing systems, such as embedded, hardware devices/sensors, or mobiles which are generally outside the scope of this survey, as we mainly focus on software systems. However, we briefly mention some of the mobile devices studies that are closely related and have aimed to replicate the efforts in software logging research in Section 2.5.5. Lastly, Figure 2.5 summarizes a taxonomy of categorization of modern-day logging research in our survey, and we leverage the classification in this figure to divide the logging research into subtopics and study them in the upcoming sections.

**Finding 1.** *Based on the taxonomy and our literature review, the logging research is spread through twelve categories (topics): ❶ costs and benefits of logging, ❷ logging practices, ❸ logging progression, ❹ log-related issues, ❺ log printing statement automation, ❻ log maintenance and management, ❼ log parsing, ❽ log-based anomaly detection, ❾ log-based runtime behavior analysis, ❿ log-based performance, fault, and failure diagnosis, ⓫ log-based user, business, security, and code-coverage analyses, and ⓬ emerging applications of logs.*

## 2.4 RQ2: What are the publication trends based on venues, topics, and years?

For this RQ, after categorizing the **103** selected publications, we organize the publications based on their venues and publication years and draw high level trends. Based on our findings from the trends, we summarize some of the log-related challenges that the prior research has aimed to address at the end of this section.

### 2.4.1 Venue Trends

Table 2.2 provides a breakdown of surveyed publications per venue. Although looked for, we could not find a related work in TOSEM [5]. The majority of the research in this field is published in EMSE, ICSE, TSE, JSS, and ASE. We suggest the authors of future publications consider the following venues in Table 2.2 for submitting their works, and

[5] ACM transactions on Software Engineering and Methodology.

consider the number of related references that their work aligns with in that venue. This ensures their works will receive higher visibility and a thorough comparison with the prior work.

### 2.4.2 Topic Trends

Figure 2.6 shows the percentage of publications per each topic. Overall, we have divided the logging research into twelve subcategories (*i.e.*, topics), which we will review throughout this survey. Table 2.19 (Section 2.8) lists the topics and provides the related references for each one. The table serves as a quick-access guide to review the research happening in each topic. Based on our analysis, the top-5 active and popular research topics in the field of logging based on the number of publications are: **1) log mining for anomaly detection**, **2) log printing statement automation**, **3) log mining for performance and failure diagnosis**, **4) log maintenance and management**, and **5) log parsing**.

### 2.4.3 Year Trends

Figure 2.7 illustrates the number of conference (blue), journal (orange), and archived (gray) publications per year till May 31, 2021. The upward trend on the plot suggests the continuous and growing interest of the research community to explore various dimensions of logging research. The publications are from both academia and industry such as Microsoft [117, 346, 62], Twitter [187], Huawei [151], RIM [277], and others [250]. Additionally, there are valuable research from the synergy between academic researchers and industry teams which further emboldens the efforts by bringing real-world industry experiences [104, 151, 208]. As such, we foresee the research in this area will continue to grow and foster in the upcoming years as there are interesting and promising trends for future research, explained in Section 2.6.

### 2.4.4 Logging Challenges

Based on the knowledge gained throughout the survey, we summarize the challenges that the prior literature has aimed to tackle in the following. As a result of a lack of well-accepted standards and guidelines for logging practices [117, 250, 78], currently, developers mostly rely on their personal experience or intuition to perform their logging decisions. However, for this manual process, *i.e.*, developers inserting logging statements into the source code, to lead to effective logging practices, we are facing four main challenges:

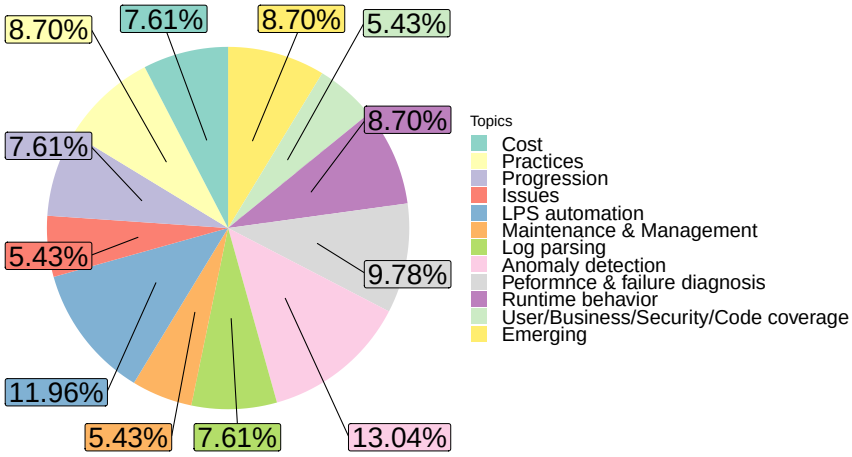



Figure 2.6: Percentage of publications in each topic.

1. The first challenge is **where-to-log**, which is the decision of selecting appropriate logging points. Logging statements can be placed in different locations of interest in the source code, such as inside *try-catch* block, *function return value*, *etc.* Although log statements provide valuable insight into the running system's state, they are I/O intensive tasks and excessive logging can incur performance and maintenance overhead [104, 339]. Consequently, developers are often faced with the challenge of making an informed decision for *where-to-log* in order to avoid introducing unjustified performance degradation and maintenance overhead.

2. The second challenge, **what-to-log**, concerns with what information to include in the log message. As explained in Figures 2.1 and 2.2, the log statement description provides a brief context of the execution and the internal variables provide more insights into the dynamic context of the current execution state. Therefore, the logging description and logged variables should satisfy their purpose and be clear and informative about the current state of the program. The logging description should also stay up-to-date and in-sync with the feature code updates, as some developers fail to update the logging statements as feature code changes [169, 280].

3. The third challenge is **how-to-log**, which concerns with how the logging code, as a subsystem, combines with the rest of the software system. As the logging code is intertwined across different source code modules, some prior researchers have suggested modularizing the logging code, as an independent subsystem, which becomes

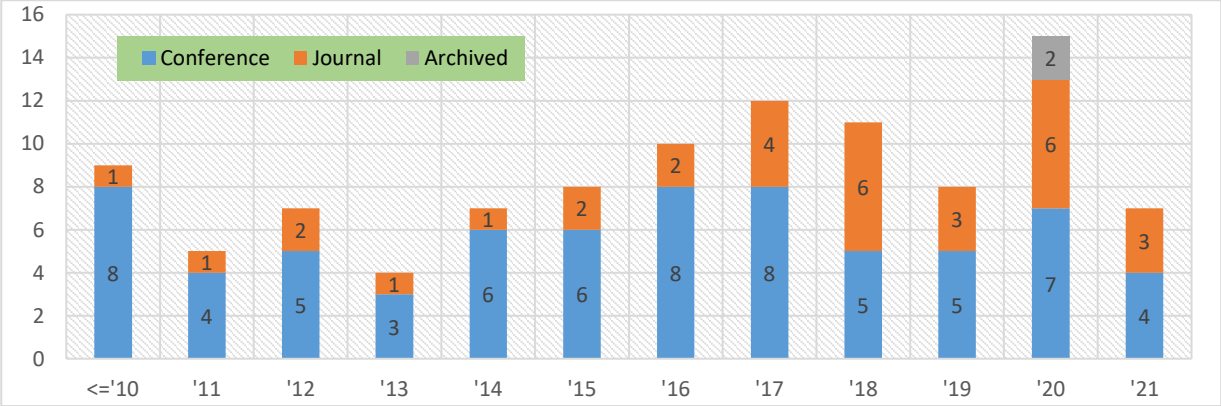



Figure 2.7: Number of publications per year divided into conference and journal categories. Publications up to May 31, 2021 has been observed at the time of this survey.

compiled into the feature code in the later stages of the system release [164, 18]. Nevertheless, many industrial and open-source software projects still tend to mix the logging code with the feature code [250, 79, 325, 78]. As a result, maintaining and developing high-quality logging code as the feature code evolves remains challenging and crucial to the overall quality of the software.

4. The fourth, and also the most recent challenge, discusses the question of **whether-to-log**, which concerns with dynamically adjusting the degree of logging in response to the runtime requirements. For example, if a suspected anomaly is detected, the logging platform can enable more detailed logging, and if the system is acting normal, it minimizes the number of logs to lessen the overhead.

Figure 2.8 summarizes the logging practices challenges. In the following, we briefly review the related research concerning each challenge. Later on, we will revisit these research efforts in more detail in their related section in the survey.

### Where to log

The research in this area is interested in finding appropriate logging points. One approach is to analyze the source code and look for specific types of code blocks, *i.e.*, unlogged ex-

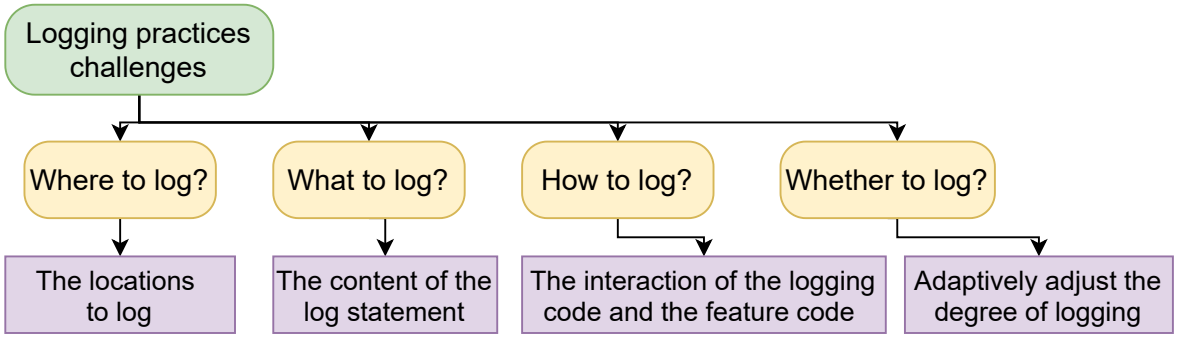



Figure 2.8: Logging practice challenges.

ception code blocks, and insert logging statements inside them [324]. Other log placement objectives, such as disambiguating execution paths [339], minimizing the I/O and performance overhead [104], and feature extraction and learning approaches [117, 346] also exist in the literature.

### What to log

The research in this area is concerned with with what the content of logging statements should be to make the logs more effective for future system observation purposes, such as debugging and failure diagnosis. Possible approaches include *automatically adding variables* that can clarify execution paths to the LPSs [326, 209]. Additionally, concerns on the content of logging statements arise when developers neglect to update the LPSs as the related feature code is updated [169], which is also a common problem in *source code comments* [293].

### How to log

This avenue is concerned with the problem of how to develop and maintain a high-quality logging code. Additionally, it pays attention to the interaction and integration of the logging code as a subsystem with the rest of the software system. Paradigms such as Aspect Oriented Programming (AOP) [164] aim to look at log statements as a submodule of the software which is separate from the feature code, and unifies with the rest of the software at a later time during the development process. However, the current logging practices in the industry and open-source projects commonly fuse the logging code inside

the feature code. Logging libraries and utilities [81] also take part in organizing the logs and improving their formatting and quality. As such, enhancement of logging libraries can positively impact on ensuring *how we log*.

#### Whether to log

One approach to tackle the challenge of the number of logs is to enable dynamic filtering of logs during runtime [104, 229]. Enablement of this paradigm would allow to take the pressure off the developers and enable more conservative addition of log statements in the source code, without being concerned about the overhead. Thus, depending on the program state, logs are dynamically discarded or collected if they serve any online or postmortem analysis purpose.

**Finding 2.** *The top five research topics for logging research are:* ❶ *log mining for anomaly detection,* ❷ *log printing statement automation,* ❸ *log mining for performance and failure diagnosis,* ❹ *log maintenance and management, and* ❺ *log parsing.*
**Finding 3.** *The top five publications venues for logging research are:* ❶ *Empirical Software Engineering (EMSE),* ❷ *International Conference on Software Engineering (ICSE),* ❸ *Transactions on Software Engineering (TSE),* ❹ *Journal of Systems and Software (JSS), and* ❺ *Automated Software Engineering Conference (ASE).*
**Finding 4.** *Due to the challenges associated with logging, i.e., what, where, whether, and how to log, there exists a continuously growing interest in log-related research and the prior work is published in top-ranked venues in yearly basis (Figure 2.7 and Table 2.2).*

## 2.5 RQ3: How the research in each topic can be systematically compared with their approaches, pros and cons?

In this section, we review the available literature in each category of logging research and provide a comparative analysis.

### 2.5.1 Category A: Logging Cost and Benefit Analysis

Although logs are useful and provide insight into the internal state of the running software, they also impose inherent costs on different subsystems of a computer system. We can

assess the costs and benefits of logging both *quantitatively* and *qualitatively*, which we review in the following.

### Quantitative assessment

Quantitative assessment for benefits of logging measures to what extent logging improves a specific debugging task. For example, Yuan et al. [324] observed the benefits of improved logs, as they contributed to ∼60% faster diagnosis time when compared with the original logging statements, *i.e.*, prior to the enhancement. Log associated overheads can be also evaluated quantitatively [104, 329]. Table 2.3 summaries **logging cost breakdown** on various subsystems of a computing system [104], including I/O, storage, CPU, and memory. For example, one approach [339] to simplify and **measure the slowdown** caused by logging statements is to calculate the number of times ($n$) each log statement is being executed and multiply that with the overhead of a single log statement execution ($l$), *i.e.*, $n \times l$. Other research efforts in this area have measured the **overhead of Linux security auditing through log messages** by enabling and disabling audit logging [329], and statistically **mapped logging statements to the performance of the system**, *i.e.*, CPU usage [317, 318].

### Qualitative assessment

In contrast to quantitative metrics for measuring logging overhead, *e.g.*, system slowdown or I/O cost, qualitative approaches aim to understand the underlying trade-offs from developers' perspectives through surveys or questionnaires. A developer survey at Microsoft [104] uncovered main overheads associated with logging, from developers' perspective, as listed as *"Details"* in Table 2.3. Developers were also inquired about the methods they use for containing the logging overhead for large-scale online service systems. They mentioned a variety of methods to limit the logging overhead such as **adjusting the logging verbosity level** (93% of developers have applied this approach), **manual removal of unnecessary logs** (64%), and **periodic archiving of log files** to save disk space (43%). Additionally, this study observed the lack of a cost-awareness guideline during log instrumentation. Some developers often had little idea about the logging overhead when they planned to add new logging statements to the source code. Thus, developers require to be more mindful in adding logging statements in scenarios such as a *for-loops*, which iterates a large number of times and could cause high overhead, especially on CPU, I/O, and storage throughput. A recent study [192], **qualitatively** examined logging cost and benefits from developers' perspectives. One qualitative measure of the logging cost, *i.e.*,

| Overhead | Details |
|---|---|
| Disk I/O bandwidth | Logging causes additional I/O bandwidth (BW) which may interfere with the required I/O BW for the system's core functionality. The BW requirement by enabling all logs (*i.e., verbose* level vs. *medium* level) can become significantly higher than the presumed BW. For example, in [104], the extra BW by enabling all logs is 8MB/s, which, however, should have been ≤200KB/s. |
| Storage | As the logging BW increases, OS might slow down, and other processes that require disk space and BW may crash, and even the logging subsystem could crash. Additionally, more logging requires more storage space. |
| CPU | As the CPU usage of the logging subsystem is increased, service to other processes is slowed down. Once the CPU usage of logging goes up to double digits, the slowdown of the other processes becomes significantly noticeable. Ding *et al.* [104] recommend an overhead of 3-5% as the CPU usage upper bound for logging. |
| Memory | Developers noticed unexpected increases in memory usage of the logging subsystem, which was the root cause of one service incident. Additionally, memory leakage of the logging system caused days of effort in debugging. |

Table 2.3: System's performance overhead associated with logging.

too much logging, causes **noisy log files** which hinders program comprehension and results in strenuous log file analysis. In contrast to costs associated with logging, main qualitative benefits of logging communicated by developers include **the capability to diagnose runtime failures with logs**, **using logs as a debugger**, **user/customer support**, and **system comprehension**. Table 2.4 summarizes the research in Category A.

| Reference - Approach | Results | Type | Pro | Con |
|---|---|---|---|---|
| Yuan *et al.* [324] - Conservatively adds log statements to the source code while aiming to minimize the introduced execution overhead. | This study categorized seven generic patterns of error sites based on the study on 250 failures, such as exceptions, function return errors, *etc.* | Quan./ Qual. | Errlog provides three different levels of configurable logging overhead. | Focuses on Error Logging Statements (ELS). |
| Zeng *et al.* [329] - Measures the overhead of Linux security auditing through log messages. | The authors measured up to 5% performance overhead when the audit logging is enabled. | Quan. | Proposes an adaptive approach to reduce the overall system overhead from 5% to 1.5% | Reduced auditing might result in a lower level of security protection for the system. |
| Ding *et al.* [104] - Surveys engineers in Microsoft and applies a constraint solving-based method to calculate an optimal logging placement. | Maximizes extracted runtime information and, concurrently, minimizes the I/O and performance overhead. | Quan./ Qual. | Two levels of filtering, *i.e.*, local and global filters, to discard less-informative logging messages and simultaneously keep important messages. | Curtailed to performance analysis of logs, and falls short for logs recording error and failure information. |
| Li *et al.* [192] - Studies developers' logging considerations when it comes to the costs and benefits associated with logging. | Main benefits of logging communicated by developers include: *diagnosing runtime failures*, *using logs as a debugger*, *user/customer support* and *system comprehension*. | Qual. | Survey of 66 developers and a case study of 223 logging-related issue reports from the issue tracking systems. | Limits to open-source projects and closed-source projects might evaluate differently on their logging costs and benefits. |
| Yao *et al.* [318] - Introduces a statistical approach to map logging statements to the performance of the system, *i.e.*, CPU usage. | If the performance model's prediction error is noticeably impacted, it implies that the modified log helps to model the CPU usage properly. | Quan. | The approach finds and suggests removing insignificant log statements. | Logging statements that are not covered by the performance tests cannot be identified by this approach. |

Table 2.4: Logging cost and benefit research - Topic (A). 'Type' shows *qualitative*, *quantitative*, or both.

### 2.5.2 Mining Log Printing Statements

There has been a significant body of research aiming to mine, understand, and characterize various source code logging practices [117, 325, 79, 78]. Because, intuitively, understanding previously applied logging practices is the gateway to help developers improve their current logging habits. Thus, to derive the in-the-field LPS practices, the first step is to mine the source code's logging statements and extract useful insight and observable patterns. Consequently, there are two broad classes of prior studies that have sought after understanding and mining of the logging practices in both industry and open-source projects: 1) **mining logging code**, and 2) **mining log files**. We review the research for mining logging code in this section and mining of log files in Section 2.5.4.

Mining log printing statements (LPSs) in the source code principally focuses on understanding how developers insert LPSs in to the source code and how logging code evolves over time to gain insight into the common logging practices. Figure 2.9 shows the categorization of research for logging code mining of software projects which is divided to research on **logging practices (Category B)**, **logging code progression (Category C)**, and **logging-code issues (Category D)**. In the tables that follow, our convention is that *pro* signifies an *advantage*, or an *improvement*, and *con* signifies a *limitation*, *room for improvement*, or an *avenue for future work*. This section includes research in Categories *B, C, and D,* that we review in detail.

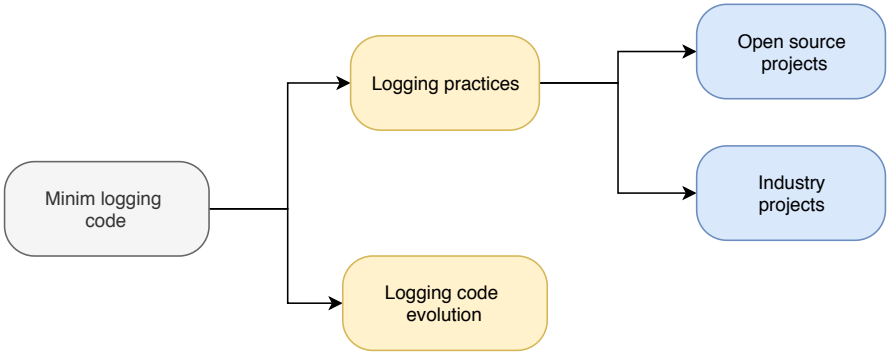



Figure 2.9: Logging code mining research in subcategories.

#### Category B: Logging Practices

Mining logging practices aims to gain insight into the current logging habits of developers both in open-source and industrial proprietary software projects.

| Reference - Aim | Experiments | Results | Pro | Con |
|---|---|---|---|---|
| Yuan *et al.* [325] - Study and characterize logging practices in four open-source C/C++ based projects. | Four software projects: *Apache httpd*, *OpenSSH*, *PostgreSQL*, and *Squid*. | Observes ten findings and their implications that software logging is pervasive and developers spend significant time maintaining logging code. | Provide a simple checker to detect verbosity level inconsistencies. | A follow-up study observed contradictory findings in some cases [79]. |
| Fu *et al.* [117] - Conducts source code analysis on two software systems at Microsoft, to categorize logged and unlogged snippets. | A questionnaire and a decision-tree classifier to detect whether a code snippet requires a logging statement. | The research uncovers five categories of logged code snippets, including *return-value-check* and *exception-catch* snippets. | Extracts contextual features and proposes a decision-tree classifier, which can detect whether a code snippet requires a logging statement. | Logging categories can be broken down further into subcategories. |
| Pecchia *et al.* [250] - Studies the logging practices on a critical industrial software at Selex ES. | Experimented with the software at Selex ES in three product lines, *i.e.*, middleware (MW), business logic (BL), and human-machine interface (HMI). | The study uncovers three main reasons for logging in the industrial domain: state dump, execution tracing, and event reporting. | Observed logging is highly developer-dependent, and company-wide **log policies** and **guidelines** are needed. | The study is limited to a very particular closed-source software system, and the findings might not generalize to software in other application domains. |
| Shang *et al.* [279] - Explores the relationship between logging characteristics and the code quality. | A case study on four releases of Hadoop and JBoss projects. | Logging characteristics provide a strong indicator of post-release defects, *i.e.*, files with more logging statements have a higher rate of post-release defect compared to the files without logging. | Developers' code improvement efforts should focus more on the source code files with high logging density or high rate of log churn. | The study cannot establish a causal relationship, *i.e.*, there might be a large portion of defects not captured due to not being logged extensively. |
| Chen and Jiang [79] - A replication work of Yuan *et al.*'s work [325] on 21 Java projects. | 21 open-source Java projects in three different domains: *server*, *client*, and *supporting components*. | Similar findings as [79] regarding logging pervasiveness and that developers' significant amount of time spent on maintaining the logging statements. | A high portion of code updates are for improving the quality of logs and contrary to [79], this research finds developers spend more time fixing reported failures when log messages are present. | Contradictory findings compared to the prior work [79] raises the concern of how useful the findings are, and if logging practices are project, programming language, and domain dependent. |
| Zhi *et al.* [342] - Conducts an exploratory study on the logging configuration practices and how they evolve over time. | 10 open-source and 10 industrial java projects in various domains and sizes. | The research's main findings show that current practices of logging configurations concerns with logging management, logging storage, logging formatting, and logging-configuration quality. | Provides a simpler checker to statically analyze and detect log configuration issues. The authors found some issues on open-source projects by applying the checker. | Further research to improve the quality of logging configurations is required to detect and resolve logging configuration smells. |
| Chen *et al.* [81] - Studies logging utilities (LUs) usage in Java project. | Over 11,000 projects and 3,850 Java LUs (e.g., SLF4J [29]) from GitHub. | With a heuristic-based technique, the study observed a positive correlation between the size of the project and the complexity of LUs. | Some projects still use multiple LUs to bring in more flexibility, and, additionally, support and enable logging behavior of the imported packages. | Currently, configuring different LUs is a manual and error-prone task. Thus, error-free and automatic checkers and techniques to configure LUs are required. |

Table 2.5: Logging practices research - Topic (B).

**Open-source projects.** Prior work in this category includes empirical studies to characterize current logging practices in open-source projects such as Apache Software Foundation (ASF) [20] projects [325, 79]. Other works [79, 279] aimed to find recurrent mistakes in the logging code and its relationship to overall source code quality. Another tread of research [81, 342] has examined logging configurations, libraries, and utilities.

**Industry projects.** Similar to the open-source software, software logging is a widely adopted practice in industry projects. Fu et al. [117] conducted a study on logging practices of two software systems at Microsoft, and Pecchia et al. [250] examined application-critical software logging practices at Selex ES.

Table 2.5 summarizes and compares the research on mining of the logging source code for open-source and industrial projects. In sum, logging is a **pervasive convention** in various software domains (*e.g.*, server, client, and support applications) and developers utilize **various logging practices** and spend a **significant amount of time** updating logging statements.

### Category C: Logging Code Progression

So far, we have discussed the research investigating logging practices in both open-source and industry projects. Prior research has also studied the progression (*i.e.*, evolution) of the logging code in software projects. Progression means that how logging code changes over time. Prior studies have concluded that logging code **evolves significantly** (*i.e.*, high churn rate), even at a higher rate than the *feature code* over the lifespan of the software development [325, 277, 339, 171, 170]. Additionally, many projects go through logging library migrations throughout their lifetime [169], and research has proposed tools to predict likely logging code revisions, *e.g.*, LogTracker [197, 196]. Table 2.6 provides additional details and summarizes the research on the progression of the logging code.

### Category D: Log-related Issues

The extensive usage of logs comes with mistakes, improper, and not well-thought logging practices, which results in logging issues and low-quality logging statements. Some of the research in this thread overlaps with *logging practices* and *logging code progression*, as some of the logging issues are uncovered during the examination of logging practices and their evolution. Yaun *et al.* [322] presented a characteristic study on real-world failures in distributed systems, and observed that the majority of failures print explicit failure-related log messages which can be used to replay (*i.e.*, recreate) the failures. However, recorded

| Reference - Aim | Experiments | Results | Pro | Con |
|---|---|---|---|---|
| Shang *et al.* [277] - Explores the progression of logging code in execution (*i.e.*, log files) and source code levels. | Two open-source *(Hadoop and PostgreSQL)* and one industrial *(EA)* software projects. | The logging code changes at a high rate across versions, which might break the functionality of log processing applications (LPA). | Developers could avoid the majority of the logging code modifications through better logging designs. | The broad range of the avoidable logging code changes raises the concern of if the observed values are software system dependent. |
| Kabinna *et al.* [169] - Studies the logging library migrations in Apache Software Foundation (ASF) projects. | Studies 223 ASF projects with their issue tracking systems in JIRA. | As more flexible logging libraries with additional features emerge, many ASF projects have undergone logging library migrations or upgrades. | Although adding more flexibility and performance improvement are cited as the primary drivers for logging library migrations, performance after library migration is rarely improved. | A questionnaire survey from developers involved in logging migration efforts can bring additional value and more insight into their rationale behind the logging updates and best practices. |
| Kabinna *et al.* [170, 171] - Investigates the stability of logging statements over time, *i.e.*, whether a logging statement will go under revisions in the future. | Four open-source projects: *Liferay*, *ActiveMQ*, *Camel* and *CloudStack*. | A significant portion of logging statements change throughout their lifetime, and factors such as file ownership can affect the stability of logging statements. | Developers of LPAs should rely on more stable logging statements for designing their tools. | The research considers only the first change of logging statements. However, the already changed logging statements might become more stable after going through modifications and prior fixes. |
| Li *et al.* [197, 196] - Studies the co-evolution process of logging statements as bug fixes and feature code updates are committed. | 12 open-source projects in C/C++ language from various domains, including *Httpd*, *Rsync*, *Collectd*, *Postfix*, and *Git*. | Proposes LogTracker, a tool that proactively predicts log revisions by correlating the rules learned from historical log revisions, *e.g.*, the logging context, and the feature code. | Utilizes code clones to learn log revision rules with the insight that semantically similar codes will likely require similar logging revisions. | The tool can only guide log revisions for codes that share similar logging context, and the percentage of these revisions is not substantial. |
| Rong *et al.* [265] - Investigates the status of developers' *intention* and *concerns* (I&C) on logging practices. | Developers' interviews and code analysis on three industrial software projects. | Major gaps and inconsistencies exist between the developers' I&C and real log statements in the source code. | For reasons such as *lack of supporting facilities* and *the version evolution of source code*, the developers' I&C are poorly reflected in the log statements. | Only missing log statements are considered as inconsistencies, and unnecessary log statements are not evaluated. |

Table 2.6: Logging code progression research - Topic (C).

| Reference - Aim | Experiments | Results | Pro | Con |
|---|---|---|---|---|
| Yuan *et al.* [322] - Presents a characteristic study on real-world failures in distributed systems to understand how faults evolve to user-visible failures. | 198 user-reported failures that occurred on *Cassandra*, *HBase*, *HDFS*, *Hadoop MapReduce*, and *Redis*. | The majority of failures print explicit log messages which can be used to replay (*i.e.*, recreate) the failures. However, the recorded log messages are noisy, which makes the analysis of logs tedious. | Provides a simple rule-based static checker, *Aspirator*, to detect the location of the code bug patterns, including log-related issues. | The study limited to a set of data-intensive systems in their production quality, *i.e.*, not during the development phase. |
| Shang *et al.* [280] - Utilizes development knowledge [118], *e.g.*, JIRA tickets [24] to understand the intention of log statements. | 300 randomly sampled logging statements, and manually examining the email threads in the mailing list for three open-source systems: *Hadoop*, *Cassandra*, and *Zookeeper*. | Identifies five categories of information that practitioners often look for to understand in log messages: **meaning**, **cause**, **context**, **impact** of the log message, and the **solution** for the log message. | The approach can be used to identify the experts for a particular log line and seek their help. | Development knowledge is considered for log lines at the method level. The higher the level, the more development knowledge that can be attached, but the more overwhelming such attached knowledge might become. |
| Chen and Jiang [78] - Characterizes anti-patterns (AP) (*i.e.*, recurrent mistakes) in the logging source code. | 352 log changes from three systems: *ActiveMQ*, *Hadoop*, and *Maven*. | Finds six different anti-patterns in the logging code, such as wrong log levels and logging nullable objects, and proposes a tool, *LCAnalyzer*, to detect anti-patterns. | The approach learns anti-patterns from how developers fix the defects in their logging code. | The work detects APs based on the independent historical changes to the logging code and falls short in detecting APs in cases that there has not been an update to the logging code. |
| Hassani *et al.* [142] - Studies log-related issues for open-source software projects. | 563 log-related JIRA issues from *Hadoop* and *Camel* projects. | As per authors findings, among the most common logging code issues are: *1) inappropriate log messages*, *2) missing logging statements*, *3) inappropriate log verbosity levels*, and *4) log library configuration issues*. | Developed a tool to detect incorrect log verbosity levels based on the words that appear in the logging statement's description. | Log-issue checkers are threshold-dependent and in some cases result in a low number of detected issues. |
| Li *et al.* [202, 201] - Studies issues with duplicate logging statements, which are logging statements that have the same static text messages. | 4K duplicate logging statements in five open-source projects: *Hadoop*, *CloudStack*, *Elasticsearch*, *Cassandra*, and *Flink*. | Repetitive logging statement descriptions can be potential logging code smells [335], *i.e.*, a problematic duplicate logging code, which can have a detrimental or misleading effect in the understanding of the dynamic state of the system. | Uncovers five categories of duplicate logging code smells and proposes a static analysis tool, DLFinder, to automatically detect duplicate logging code smells. | The research eliminate the top 50 most frequent words when detecting log message mismatch (LM), which might cause false negatives. |

Table 2.7: Log-related issues research - Topic (D).

log messages are noisy, which makes the analysis of logs tedious. Several efforts have aimed to identify and reduce log-related issues, such as finding recurrent **logging mistakes**, *i.e.*, anti-patters [78], **adjusting verbosity levels** (sometimes back-and-forth), **adding missing variables**, **modifying static text** to fix inconsistencies [325], and finding **logging code smells** [201, 202], *i.e.*, duplicates. Hassani *et al.* [142] empirically categorized log-related issues in open-source projects, among them **inappropriate log messages** and **missing logging statement itself** (also [339]) in locations that have to be logged. The detection of logging code issues would be helpful, as developers can add revisions to logging statements, and hence improve the quality of log statements. As such, in addition to tools that automatically detect log-related issues such as DLFinder [201] and LCAnalyzer [78], future research will benefit from developing tools that can automatically *fix* log-related issues. In addition, due to the lack of proper communication with developers of large-scale software systems, practitioners, who review logs for software maintenance tasks, might encounter challenges in understanding the logging messages. Such challenges may hamper the effectiveness and correctness of leveraging logs. Therefore, utilizing development knowledge [280], in particular issue reports for log statements, *e.g.*, JIRA tickets [24], can help practitioners to better understand log messages. Shang *et al.* [280] identified five categories of information that practitioners often look for to understand in log messages: **meaning**, **cause**, **context**, **impact** of the log message, and the **solution** for the log message. The key takeaway is that leveraging development knowledge, such as issue reports and code commit information, helps in clarifying the log messages. Table 2.7 summarizes the research in Category D.

### 2.5.3 Category E: Log Statement Automation

As mentioned earlier, execution logs, which are the output of logging statements in the source code, are a valuable source of information for system analysis and software debugging. Thus, high-quality logging statements are the precursor of the effective log file mining and analysis. Conversely, low-quality LPSs result in log-related issues (Section 2.5.2), and they hinder the understanding of software problems whenever they happen. Currently, due to the *ad-hoc* nature of logging, lack of general guidelines, and because developers mostly insert logging statements based on their personal experiences, the quality of log statements can hardly be guaranteed [333]. Therefore, automated logging which aims to add or enhance log statements inside the source code either proactively or interactively is a well-motivated effort and can improve the quality of logging statements and, ultimately, result in more effective log mining tasks.

Figure 2.10 presents the log statements' research with emphasis on automated logging.

| Reference - Aim | Experiments | Results | Pro | Con |
|---|---|---|---|---|
| Zhang *et al.* [333] - Proposes *AutoLog*, which generates additional informative logs to help developers discover the root cause of a software failure. | Performs a proof-of-concept case study on Apache Hadoop Common. | AutoLog embeds a two-stage process of *log slicing* and *log refinement* of the program to narrow down the execution paths that could have led to the system's failure. | The approach narrows down the execution paths that could have led to a system's failure. AutoLog is targeted for interactive in-house development. | The program needs to be re-executed every time new log statements are added, which is time-consuming. |
| Yuan *et al.* [326] - Proposes a tool, *LogEnhancer*, to find and add useful variables to log statements. | Evaluated on a total of 9,125 log messages from eight applications in different domains, including *apache httpd*, *postgresql*, and *cvs*. | LogEnhancer is effective in automatically adding a high percentage of log variables, on average, 95.1%, that programmers manually included. | The tool performs static analysis on the source code starting from the log statement and navigates backward to find variables that are causally along the path that results in the execution of the log statement. | As LogEnhancer's improvement is limited to the existing log statements, its effectiveness diminishes if the logging statements are missing. |
| Zhu *et al.* [346] - Proposes *LogAdvisor*, which aims to provide logging suggestions for *exception* and *return-value-check* code blocks. | Two industrial software systems from Microsoft and two open-source software systems from GitHub (*SharpDevelop* and *MonoDevelop*). | LogAdvisor achieves a high *balanced accuracy*, ranging from 84.6% to 93.4%, to match developers' logging decisions, and the decision tree model achieves the highest scores. | Trains a machine learning model (*e.g.*, SVM and decision trees) to predict whether a focused code snippet requires a logging statement. | It is focused and limited to two categories of code snippets: 1) *exception snippets* and 2) *return-value-check*. |
| Lal *et al.* [184] - Introduces *LogOptPlus* tool for automated *catch* and *if* code block logging prediction. | Two open-source projects: *Apache Tomcat* and *CloudStack*. | The prediction model with random forest achieves the highest F1-score 80.70% (Tomcat) and 92.25% (CloudStack) for *if-block* logging prediction. | Applies five different learning techniques, *e.g.,* AdaBoost, Gaussian Naive Bayesian, and Random Forests achieve the highest Precision and Recall. | It is limited to specific code blocks, *i.e.*, *if-block* and *catch clause*. |
| Zhao *et al.*[339] - Introduces *Log20*, a tool that finds a placement of logging statements to minimize execution path ambiguity. | Evaluated on four open-source Java projects: *HDFS*, *HBase*, *Cassandra*, and *ZooKeeper*. | Log20 achieves a lower logging overhead with the same level of informativeness (*i.e.*, entropy) compared to existing logging statements by developers. | It applies Shannon's information theory equation to measure the entropy of the program by approximately considering all of the possible execution paths. | The approach does not consider developers' concerns and practices, does not explain the static content of LPSs, and change of workload can cause extra logging overhead. |
| Li *et al.* [194] - Analyzes log changes in open-source projects and proposes commit-time logging suggestions. | Four open source projects: *Hadoop*, *Directory Server*, *Commons HttpClient*, and *Qpid*. | Performs a manual analysis on four software systems and categorizes the changes to logging statements into four major groups: 1) *block change*, 2) *log improvement*, 3) *dependence-driven change*, and 4) *logging issues*. | Proposes a random forest classifier for each code commit to suggest whether a log change is required. The classifier's balanced accuracy for within-project suggestions is 0.76 to 0.82. | As the model is trained on prior log changes, it might miss scenarios that there are no prior logging changes to learn from. |
| Li *et al.* [193] - Determines the appropriate log verbosity level for the newly-developed logging statement. | Analyzes four open source projects: *Hadoop*, *Directory Server*, *Hama*, and *Qpid*. | Collected five categories of quantitative metrics that play important roles in determining the appropriate log level: *logging statements metrics, containing block metrics, file metrics, change metrics, and historical metrics*. Achieves AUC in the range of 0.75 to 0.81 for log level prediction. | Metrics from the block which contains the logging statement, *i.e.*, the surrounding block of a logging statement, play the most important role in the ordinal regression models for log levels. | The results show that the ordinal regression models for log level prediction are project-dependent. |

Table 2.8: Log printing statement automation research - Topic (E).

| Reference - Aim | Experiments | Results | Pro | Con |
|---|---|---|---|---|
| Jia *et al.* [166] - Proposes, *SmartLog*, which is an intention-aware error logging statement (ELS) suggestion tool with two intention models: IDM and GIDM. | Experiments on six open-source projects in C/C++: *Httpd*, *Subversion*, *MySQL*, *PostgreSQL*, *GIMP*, and *Wireshark*. | It improves on the Recall values (average of 0.61) and achieves higher scores compared to LogAdvisor [346] (average of 0.45) and Errlog [324] (average of 0.18). | This work improves on prior work by going beyond code patterns and syntax features, and considers source code intentions, *i.e.*, semantics. | This work is limited to ELS prediction, *i.e.*, *exception* and *function return-value* logging. |
| Anu *et al.* [54] - Proposes a method to make the logging level decisions by understanding the logging intentions. | Four open-source software projects: *Hadoop*, *Tomcat*, *Qpid*, and *ApacheDS*. | It reaches AUC values higher than 0.9 in log level prediction. The approach extracts the contextual features from logging code snippets and leverages a machine learning model (*i.e.*, a random forest model) to automatically predict the verbosity level of logging statements. | As a proof of concept, the authors also implement a prototype tool, *VerbosityLevelDirector*, to provide guidance on log verbosity level selection in focused code blocks. | The work is limited to focused code blocks: *exception handling blocks* and *condition check blocks*. |
| Liu *et al.* [209] - Presents an approach to recommend the variables to include in logging statements. | Evaluates on nine open-source Java projects: *ActiveMQ*, *Camel*, *Cassandra*, *CloudStack*, *DirectoryServer*, *Hadoop*, *HBase*, *Hive* and *Zookeeper*. | The approach first learns "rules" from existing logged code snippets by extracting contextual features with deep learning recurrent neural networks (RNN). The approach outperforms five baselines, including random guessing and IR methods in log variable prediction. | The tool provides a ranked list of variables that probably are required logging to the developer. | The method only considers the code preceding the logging statement. As such, extending this approach to include the code succeeding the logging statement can improve on logging variable recommendation. |
| Kim *et al.*[175] - Proposes an approach to verify the appropriateness of the log verbosity levels. | 22 open-source projects from three different domains: *message queuing*, *big data*, and *web application server*. | Applies semantic and syntactic features and recommends a new log level in case the current level is deemed inappropriate. It reaches 77% precision and 75% recall in log level validation. | Creates *domain word model* from all of the log messages in application domains, which enables knowledge sharing between different projects. | In some cases, the appropriateness of log levels is dependent on developers' opinions and is quite arguable. |
| Gholamian and Ward [123] - Proposes a log-aware code clone detection (LACC) approach for log suggestions. | Three open-source Java projects: *Tomcat*, *Hadoop*, and *Hive*. | Performs an experimental study of logging characteristics of source code clones and observes that code clones match in their logging behavior. Achieves 90% accuracy in log location prediction. | It applies source code features and machine learning methods to detect log-aware code clones for log statement prediction. | The approach can only suggest logs for code snippets that can find their clone pairs in the software code base. |
| Li *et al.* [200] - Discusses the locations that need to be logged, and proposed a learning approach to provide code block level logging suggestions. | Seven open-source systems: *Cassandra*, *Elasticsearch*, *Flink*, *HBase*, *Kafka*, *Wicket*, and *ZooKeeper*. | The authors discover six categories of logging locations in different types of code blocks from developers' logging practices. It achieves balanced accuracy of 80.1%) using syntactic source code features. | Utilizes a pipeline of word embedding, RNN layer, and a dropout layer in its deep learning model for log location prediction. | The achieves acceptable prediction by leveraging syntactic information only. Additional studies are required to combine syntactic and semantic features of the source code blocks. |
| Cândido *et al.* [72] - Proposes a log suggestion approach based on machine learning methods. | An enterprise software, Adyen, and 29 Apache projects. | The authors extract source code metrics from methods and evaluate the performance of five different learning approaches on log suggestions. The best performing model achieves 72% of balanced accuracy on Adyen's log statements set. | Performs a study on 29 Java projects and leveraged learning transfer to generalize to an industry project. | The applied transfer-learning approach shows a lower performance when trained on open-source projects and tested on Ayden enterprise project. |
| Li *et al.* [203] - Proposes a deep learning approach for log level prediction with an ordinal-based output layer. | Nine large-scale open-source projects: *Cassandra*, *ElasticSearch*, *Flink*, *HBase*, *JMeter*, *Kafka*, *Karaf*, *Wicket*, *Zookeeper*. | The authors initially perform a manual study and categorize five different logging locations. The model trained with syntactic features achieves an average AUC of 80.8%. | Their findings infer that the log levels that fall far apart on the verbosity scale manifest different characteristics. | Log levels that are closer in order, *e.g.*, warn and error are more difficult to distinguish with this approach. |

Table 2.8: Log printing statement automation research - Topic (E) (continued).

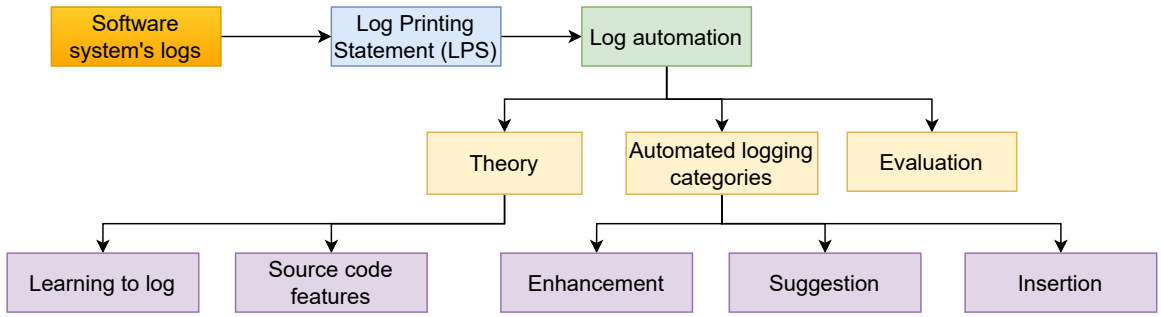



Figure 2.10: Logging research with the emphasis on logging code automation research, Category E.

As per this figure, we review the theory behind the automated logging, automated logging approaches, and how they are evaluated. Prior studies have suggested creating and utilizing statistical models from common logging practices, and to learn logging heuristics from experience, and use them to provide **new logging suggestions** or **enhance the already existing LPSs**.

### Log Automation - Motivation and Theory

One of the common approaches for log automation is the application of machine learning methods to predict whether a code snippet needs a logging statement by training a model on a set of logged code snippets, and testing it on a new unlogged code set, *i.e.*, supervised learning. In this section, we first review the background and theory for machine learning methods and continue with automated logging approaches.

**Motivation.** With the ever-increasing size of software systems, it is most likely that a single developer is in charge of developing only a small subsystem of the whole software system. Under this situation, making wise logging decisions becomes quite challenging as developers do not have full knowledge of the whole system [346]. As logs are quite pervasive and useful for system maintenance [150], if the logging decisions can be learned automatically, a log suggestion tool can be constructed to help developers making better decisions. Ultimately, such a tool can increase the quality of logs and save developers time.

**Learning to Log.** The idea of *learning to log* is to construct a machine learning (ML) model that can learn common logging practices and provide logging suggestions to the developers or directly make logging decisions and insert logging statements into a newly-developed source code snippet. A typical learning to log tool [346] is outlined in

Figure 2.11. The log learning steps are: **1)** *code collection from repositories*, **2)** *labeling the collected source code*, **3)** *feature extraction and selection*, **4)** *feature vectors and model training*, and finally **5)** *logging enhancement, suggestion or automatic insertion*. Based on this platform, once the *training phase* is completed, during the *testing phase*, the learning model decides whether a new code snippet requires a logging statement by extracting its features and feeding it to the ML model, and observing the model's output. Learning algorithms apply a wide range of techniques such as: pattern or rule-based [324, 326, 91, 339], machine learning such as Naive Bayes, Bayes Net, Logistic Regression, SVM, and Decision Trees [346], Random Forrest [194, 72], Ordinal Regression [193], and most recently, Deep Learning [123, 200, 209, 203].

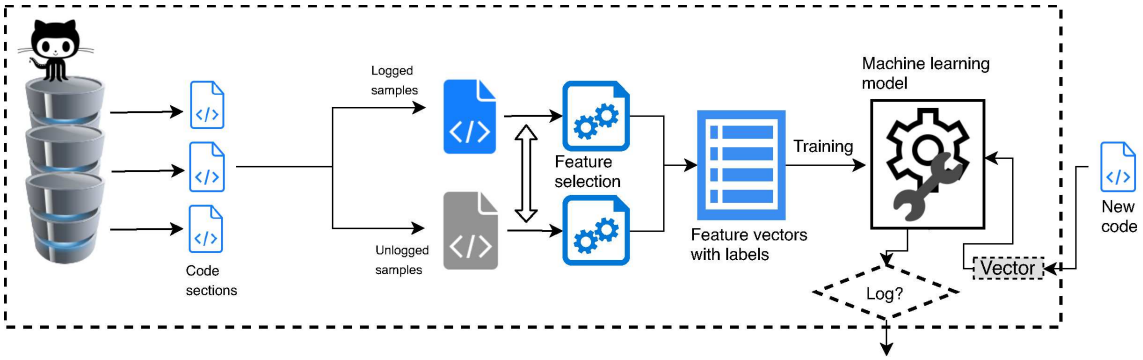



Figure 2.11: *Learning to log* platform.

### Source Code Feature Formulation

In order to be able to learn and predict log statements, prior research [346, 123] have proposed to define related source code features and utilize them for predicting whether a code block requires a log statement. Source code features can be **structural** (type of the code blocks, *e.g.*, *catch clause, if-else*), **functional** (e.g., metrics such as *code complexity, dependencies, fan-in, and fan-out*), **contextual** (*e.g.*, variables and keywords in the code snippet), and source code **semantic** features [200, 166], *i.e., what the code snippet is trying to do*. What category of features to select and how well they can distinguish the logged and unlogged code snippet is an active research topic [200, 123, 346, 186, 166]. Additionally, the logging automation research has benefited from leveraging the findings in adjacent software tasks such as source code clone detection [270], and code commenting [161] for feature selection as the idea is that similar code snippets should follow similar logging patterns. Figure 2.12 shows a log prediction platform based on similar code snippets

(*i.e.*, clone pairs), which are then later utilized for log prediction. Source code features are extracted from *method definitions* with logging statements. Then, once the machine learning model is trained and clone pairs are extracted, they are leveraged for log location prediction.

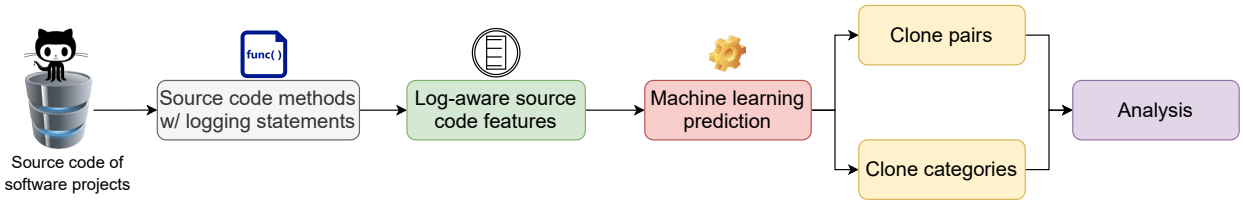



Figure 2.12: Log prediction with source code features and code clones.

### Automated Logging Categories

Figure 2.13 highlights the research in this area categorized into three subtopics: *log enhancement, log suggestion,* and *log insertion*. These approaches are primarily concerned with log **location** prediction, *i.e., where to log*, and secondarily the **content** to include in the logging statements, *i.e., what to log*. Log enhancement aims to improve the quality

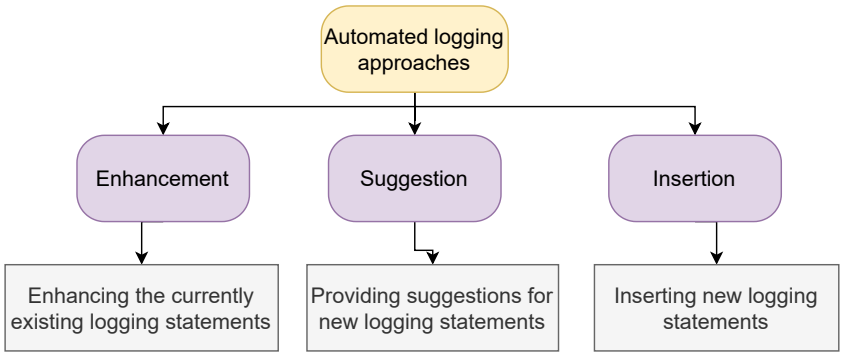



Figure 2.13: Auto logging of the software systems' source code.

of existing logging statements, such as adding more runtime context [326]. Log suggestion aims to provide suggestions for logging locations that might have been missed, and log insertion aims to proactively insert logging statements into the source code [339]. Although the goals of these approaches are similar, the intended way of their implementation can be different. For example, one practical scenario of implementing log suggestion approaches

is as IDE plugins that provide just-in-time suggestions [123]. However, the log insertion techniques are implemented as post-processing tools that scan the source code and insert logs for various criteria of interest, such as disambiguating execution paths [339] or logging catch clauses [324]. This categorization is not mutually exclusive and some of the prior work overlap in their approaches.

**Log Verbosity and Description Predictions.** In addition to predicting log location, prior research has also investigated approaches for prediction of the appropriate logging **verbosity level** for newly composed logging statements [203, 193, 142, 54, 229]. The approaches either apply some type of learning to predict the log verbosity level [54], or perform dynamic adjustment of the log level during the runtime [229]. Other research also aims to predict the description [145], or variables included in the logging statements [209, 326]. In Table 2.8, we compare and summarize the prior efforts in automated logging.

### Evaluation Metrics

After training the learning model, its performance should be evaluated during the testing phase by applying new code instances as input and find out the prediction outcome that whether or not this new code snippet requires a logging statement, *e.g.*, Figure 2.11. This is an example of a *binary classification* problem [114]. Differently, Log verbosity level prediction is evaluated as a *multi-class classification* problem [52], as generally several verbosity levels are available for the log statements, *e.g.*, *WARN, INFO, DEBUG*, *etc.* Furthermore, ordinal multi-class classification [195] considers an order between the possible prediction labels. For example, for verbosity level prediction, $WARN < INFO < DEBUG$, such that $WARN < INFO$ means $INFO$ is more verbose than $WARN$ [203]. Thus, different evaluation metrics are applied to assess the quality of learning models and their prediction accuracy. In general, the performance of a logging prediction method is evaluated by first extracting the confusion matrix.

| | | Model prediction | |
|---|---|---|---|
| | | Positive | Negative |
| Actual (ground truth) | Positive | $TP$ | $FN$ |
| | Negative | $FP$ | $TN$ |

Table 2.9: Confusion matrix for log prediction.

In Table 2.9, *"Model prediction"* values are from the learning model and the *"Actual"* values are the *ground truth*. Prior research often considers the developers' inserted logging statements as *ground truth*. To create a set of training and testing data for the machine

learning process and have a proper *ground truth* to compare with, one approach is to collect all of the code snippets with logging statements, and some samples of unlogged code, to include both positive and negative cases. Then, after deciding the train-test split and training the ML model, prior work removes the log statements from the test data. During the testing phase, the model's performance is evaluate on the test code snippets with their logging statements being removed. This way we measure how well the model can **remember** which code snippets should have and which ones should not have logging statements, compared with the developers originally-inserted LPSs. Multiple iterations of the training-testing can be applied, *e.g.*, cross-validation [71], to confirm the results. From Table 2.9, *TP* means that the model correctly predicted a code snippet that requires a logging statement, and *FN* denotes that the model incorrectly predicted that a code snippet does not require a logging statement.

Based on the confusion matrix, we can define some of the common metrics for evaluating the performance of log learning models in Table 2.10. The definitions for *Precision* and *Recall* are straightforward. In order to ensure a prediction model benefits from equally good or comparable Precision and Recall values, *F-Measure* is defined as the harmonic mean of Precision and Recall. Qualitatively, good performance of *F-Measure* implies good performance on both Precision and Recall. *Accuracy* represents correctly identified logged instances to the total number of cases. *Balanced Accuracy (BA)* is the average of the proportion of logged instances and the proportion of unlogged instances that are correctly classified. In case there is an imbalance in the data, *e.g.*, in Table 2.9, if *TN* is much larger than *TP*, *Balanced Accuracy (BA)* is widely used to evaluate the modeling results [336, 346, 194], because it avoids the over-optimism that *accuracy* might experience. Receiver Operating Characteristic (ROC) plots *true positive rate* against *false positive rate*. AUC (area under the curve) is the area under the ROC curve. Intuitively, the AUC evaluates how well a learning method can distinguish logged code snippets and unclogged ones. The AUC ranges between 0 and 1. A high value for the AUC indicates a high discriminative ability of the learning model; an AUC of 0.5 indicates a performance that is no better than random guessing [194]. BLEU [247] and ROUGE [206] scores are equivalent to Precision and Recall and are leveraged to evaluate the auto-generated text compared to the original text developed by developers. These scores have applications in evaluating the auto-generated **l**og **s**tatement **d**escriptions (LSDs), which are sequences of tokens, *i.e.*, words. For example, for a candidate LSD, $C$ and the reference LSD, $R$, BLEU measures the ratio of tokens of $C$ that also appear in $R$ (analogous to Precision), and ROUGE measures the ratio of tokens of $R$ that have appeared in $C$ (analogous to Recall). The range of values for BLEU and ROUGE is *[0,1]*, with 1 being the perfect score. These two measures combined explain the quality of the auto-generated LSDs.

| Metric | Formula | Description |
|---|---|---|
| Precision | $\frac{TP}{TP+FP}$ | The ratio of correctly identified positive instances to the number of all positive predictions. |
| Recall (a.k.a. *sensitivity*, *hit rate*, and *true positive rate*) | $\frac{TP}{TP+FN}$ | The ratio of the correctly predicted instances to the number of existing positive instances. |
| False Negative Rate | $\frac{FN}{FN+TP}$ | The ratio of false negatives to the total number of existing positive instances. |
| True Positive Rate | $\frac{TP}{TP+FN}$ | The ratio of true positives to the total number of existing positive instances. |
| True Negative Rate (a.k.a. specificity, and selectivity) | $\frac{TN}{TN+FP}$ | The ratio of true negative to the total number of existing negative instances. |
| False Positive Rate (a.k.a. fall-out) | $\frac{FP}{FP+TN}$ | The ratio of false positives to the total number of existing negative instances. |
| F-Measure (a.k.a, F-Score, F1-Score) | $\frac{2\times Precision\times Recall}{Precision+Recall}$ | Harmonic mean of Precision and Recall. |
| Accuracy | $\frac{TP+TN}{TP+TN+FP+FN}$ | Accuracy is the proportion of correctly identified logged instances to the total number of cases. |
| Balanced Accuracy (BA) | $\frac{1}{2}\times(\frac{TP}{TP+FN}+\frac{TN}{TN+FP})$ | Balanced accuracy (BA) is the average of the proportion of logged instances and the proportion of unlogged instances that are correctly classified. |
| Receiver Operating Characteristic curve (ROC) | $\frac{TPR}{FPR}$ | The plot of the true positive rate against the false positive rate. |
| Area Under Curve | $area_{under}(ROC)$ | *Area under the curve* is the area under the Receiver Operating Characteristic curve. |
| BLEU (BiLingual Evaluation Understudy) | $\frac{count\_tokens(C\cap R)}{count\_tokens(C)}*$ | Similar to Precision but for auto-generated text. The ratio of the candidate tokens (C) that exist in reference tokens (R) over the total candidate tokens. |
| ROUGE (Recall-Oriented Understudy for Gisting Evaluation) | $\frac{count\_tokens(C\cap R)}{count\_tokens(R)}*$ | Similar to Recall but for auto-generated text. The ratio of the reference tokens that exist in the candidate tokens over the total reference tokens. |

* Simplified formulas are presented. There is also weight corresponding to the size of n-grams that comes in the general formula for BLEU [247]. Similarly, for ROUGE, refer to [206].

Table 2.10: Evaluation metrics for automated log prediction.

**Examples of Metrics Used.** Li *et al.* [201] utilized *Precision* and *Recall* to calculate the performance of *DLFinder* in detecting logging code smells. Zhu *et al.* [346] used *BA* to evaluate the accuracy of *LogAdvisor*, which advises the developer if logging statements are required for a focused code snippet. Li *et al.* [194, 193, 191] used *ROC* and *AUC* to evaluate their methods in *log verbosity level prediction* and *logging commit change suggestion*. Kim *et al.* [175] used *F-Measure* to evaluate their log verbosity level recommendation approach, and Gholamian and Ward [123] utilized *Accuracy* to evaluate the performance of their log-aware clone detection approach. He *et al.* [145] leveraged BLEU and ROUGE scores to evaluate the effectiveness of the candidate log statement descriptions when compared to the developer-inserted log descriptions.

### 2.5.4 Mining Log Files

Priorly, we mentioned the purpose of logging statements added by developers is to expose valuable runtime information. The output of logging statements is written to log files, which are used by a plethora of log processing tools to assist developers and practitioners in different tasks such as software debugging and testing [53, 167], performance monitoring [317], and postmortem failure detection and diagnosis [323, 234, 291, 314]. We review log mining techniques and approaches in the following.

#### Category F: Log Maintenance and Management

As the size of logs increases, the job of methods and tools which manage and maintain logs becomes more crucial, cumbersome, and of value. For example, FLAP [198] provides an end-to-end platform for log collection, maintenance, and analysis. In the following, we review *log collections*, *log compression*, *log rolling*, and *log removal*, and Table 2.11 summarizes the research in this category.

**Log Collections.** The aim of maintaining a log collection is for auditing [188] or enabling benchmarking for different types of log analysis [149, 112, 222]. For example, Loghub [151] provides a repository of logs from various software systems, and Cotroneo *et al.* [40] have released an OpenStack failure dataset containing injected faults. The logs are used in various prior works for evaluating tasks such as compression techniques [319, 287], failure analysis [314, 323, 149] and bug detection [94]. We observe that although execution logs of different systems are conveniently available, it is difficult to find large-scale collections of labeled datasets, which are especially crucial for supervised learning of log mining tasks [311]. This is because manually labeling large datasets of logs is quite cumbersome. Thus, we see significant value in curating a database of labeled logs (*e.g.*, normal vs. anomaly/failure log records), and the development of automated log labeling techniques [315].

**Log Compression.** With the continued growth of large-scale software systems, they tend to generate larger volumes of log data every single day, which makes the analysis of logs challenging. As such, to cope and contain this challenge, developers and practitioners apply tools for compression and continuous archiving of logs [319, 208]. Hassan et al. [140] applied log compression to extract common usage scenarios. Yao *et al.* [319] studied the performance of general compressors on compressing log data relative to their performance on compressing natural language data. Their work reviews 12 widely used general compressors to compress nine log files collected from various software systems. Because log files generally benefit from higher repetition than natural text [319], there is

| Reference - Aim | Experiments | Results | Pro | Con |
|---|---|---|---|---|
| Li *et al.* [199] - Proposes a data-driven management framework by knowledge acquisition from historical log data. | Log files collected from several Windows machines in a university network. | Performs experiments on categorizing dependent and independent log events, and applies text mining techniques to categorize log messages, mines temporal data, and performs event summarization. | Provides a graphical representation of temporal relationship among events as an event relationship network (ERN) [253]. | Common categories of log messages are manually determined, which can be automated from historical data. |
| Marty [221] - Proposes a proactive logging guideline to support forensic analysis in cloud environments. | N/A (the research does not provide experimentation). | Discusses the challenges of logging in the cloud environments such as *decentralization* and *volatility* of logs. | Outlines the guidelines for when the logging is required: *business relevant*, *operational*, *security*, *compliance*. | The guideline can be expanded to include forensic timeline analysis of logs, log review, and log correlation. |
| Li *et al.* [198] - Introduces FLAP, a web-based integrated system to utilize data mining techniques for log analysis and knowledge discovery. | *Network X* event logs at Huawei Technologies. | Performs a case study and the results show the approach's applicability for different tasks, such as event summarization (graph) and root cause analysis. | It provides learning-based log event extraction and provides event summarization and visualization. | Some of the tasks, *e.g.*, root cause mining, rely on domain knowledge to manually diagnose possible problems. |
| Liu et al. [208] - Proposes a new log compression method, logzip, to allow for more effective log compression. | Five log datasets: *HDFS*, *Spark*, *Windows*, *Android*, and *Thunderbird*. | Achieves higher log compression ratios compared to general-purpose compressors, *e.g.*, *bzip2*, and can generate compressed files around half of the size of general-purpose compressors. | Performs iterative clustering with template extraction and parameter mapping and can compress in three incremental levels: L1: *field extraction*, L2: *template extraction*, and L3: *parameter mapping*. | The performance of the decompression should be also evaluated and compared with other compressors. |
| Yao *et al.* [319] - Studies the performance of general compressors on compressing log data relative to their performance on compressing natural language data. | Nine system logs, such as *HDFS* and *LinuxSyslogs*, and two natural language (NL) data, *Wiki* and *Gutenberg*. | Reviews twelve widely used general compressors to compress nine log files collected from various software systems. The observation is that log data is more repetitive than natural language, and log data can be compressed and decompressed faster than NL with higher compression ratios. | One of the findings is that general compressors perform better on small log sizes, and their default compression level is not optimal for log data. | The findings and implications of this research have not been utilized to propose a log-aware compressor. |

Table 2.11: Log maintenance and management research - Topic (F).

| Reference - Aim | Experiments | Results | Pro | Con |
|---|---|---|---|---|
| Shin *et al.* [285] - Introduces *LogCleaner*, which performs periodicity and dependency analyses for removing repetitive logs. | Two proprietary and eleven publicly available log datasets including: *CVS*, *RapidMiner*. | The approach can accurately detect and remove 98% of the operational messages and preserve 81% of the transactional log messages, and reduces the execution time of the *model inference* task from logs. | Segregates and only keeps transactional messages, which record the functional behavior of the system from operational messages of the system. | The performance of LogCleaner is heavily dependent on the quality of upstream log parser and template extraction, and requires manual analysis and domain knowledge. |
| He *et al.* [151] - Provides a repository, *Loghub*, of logs from various software systems. | Provides 17 log datasets from various application domains, including *distributed systems*, *supercomputers*, and *operating systems*. | Provides a framework for AI-powered log analysis and applies a practical usage scenario of Loghub for anomaly detection for supervised and unsupervised approaches. | Loghub datasets have been widely utilized for research both in academia and industry. | There is still a shortage of labeled datasets to facilitate the evaluation of supervised log analysis tasks. |
| Chen and Jiang [77] - Performs a survey on log instrumentation techniques. | N/A (the research does not provide experimentation). | Focuses on the three log instrumentation steps: *logging approaches*, *logging utility integration*, and *logging code composition*. | Defines four categorizes of challenges for instrumentation: *usability*, *diagnosability*, *logging code quality*, and *security compliance*. | The research can be improved by providing a connection between *logging source code* and its corresponding *log messages* in the *log files*. |
| Locke *et al.* [210] - Proposes *LogAssist* to assist practitioners with organizing and summarizing logs. | Logs from one enterprise (ES) and two open source systems: *HDFS* and *ZooKeeper*. | Groups logs into event sequences to extract workflows and illustrate the system's runtime execution paths. LogAssist shrinks the log events by 75.2% to 93.9% and the unique workflow types by 70.2% to 89.8 in HDFS and Zookeeper logs. | *LogAssist* is able to reduce the number of log events of interest to practitioners, thus it saves time and improves the practitioners' experience in log analysis. | In some cases, the searched keywords (*e.g.*, *"error"* or *"exception"*) for finding problematic log lines result in a large quantity of logs for practitioners to manually review. |

Table 2.11: Log maintenance and management research - Topic (F) (continued).

an avenue of outstanding work to develop log-aware compression techniques, that consider log characteristics in their algorithms and their parameter selections and achieve a higher compression/decompression performance.

**Log Rolling.** As log data generally grows rapidly during the system's execution [189, 319], logging libraries such as Logback [25], Log4j/2 [15], and SELF4J [29] often support the continuous rolling *i.e.*, archiving of log files as new logs become available. For example, as the size of the generated log goes beyond a user-defined value on the storage or a specific time interval has passed (*e.g.*, daily, weekly), a Log4j Appender [37] can zip, rename, and store the log with a timestamp, *e.g.*, "`logs/app-%dMM-dd-yyyy.log.gz`". Log archiving helps with the long-term maintenance and organization of logs. There is certainly room for further research on improving and automating log archiving policies and techniques.

**Log Removal.** Although logs are useful, but due to the large volume of them, they can become noisy, hard to analyze, and cause inaccuracy in log analysis. As such, prior research has aimed to detect and remove duplicate log messages [285, 92, 290]. For example, Shin *et al.* [285] introduced *LogCleaner*, which performs *periodicity* and *dependency* analyses to filter out and remove periodic and dependent log messages. In sum, the approaches that are applied for log maintenance and management facilitate automated analysis of logs, and furthermore, will yield more accurate log analysis.

### Automated Log File Analysis - Challenges and Motivation

Logs record system runtime information and are widely used and examined for assessing the systems' health and availability [150]. Traditionally, developers or operators often inspect the logs manually with keyword searches (*e.g.*, "fail", "exception") and rule matching (*e.g.*, *"grep <RegEx>"*) to find any potential problems in case of a system failure. However, manual or keyword inspection of log files becomes impractical with the ever-increasing complication of software systems because of the following reasons [150]:

❶ As current computer systems generate a massive volume of logs, *e.g.*, at the rate of 50 gigabytes per hour (around 120∼200 million log lines) on Alibaba Cloud Computing Mailing System (Aliyun Mail) [226], this makes it close to impossible to manually extract useful information from the log files and track down any system issues.

❷ The complex and concurrent nature of software systems makes it unmanageable for a single developer to comprehend the entire functionality of the system, as a single developer might be only responsible for the development of a small sub-module of the entire project. For example, hundreds of developers take part in the development

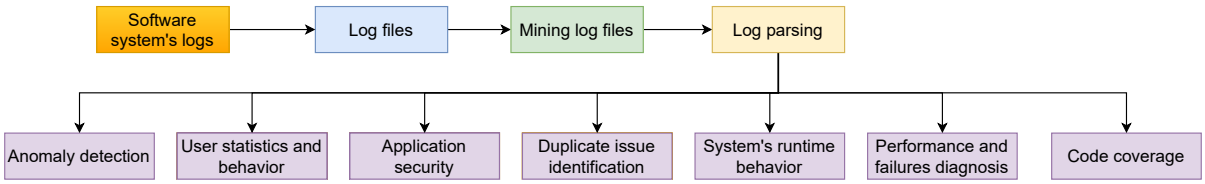



Figure 2.14: Mining of log files for different applications.

of parallel computing platforms, such as Apache Spark [16]; thus makes it quite challenging, if not impossible, for a single developer to pin down an issue from concurrent and massive log files.

❸ Parallel and distributed software systems generally apply various methods of fault-tolerant and performance optimization techniques in order to recover from a hardware failure or perform load-balancing and scheduling. For example, a resource manager daemon, *e.g.*, on a Hadoop or Spark cluster, may intentionally terminate a running application and move it to another node in the system in order to expedite the execution of that task. As a result, the traditional and manual way of searching in the log files for keywords such as *killed, terminated, failure* might not be useful and can lead to multiple false-positive cases [207] and further muddle the manual inspection.

Moreover, although automatic log analysis helps developers and practitioners significantly to speed up the process (*e.g.*, [340, 337, 314]), the automatic log analysis itself is still very challenging because log messages are usually unstructured free-form text strings, and application behaviors are often very heterogeneous and complicated [116]. As a result, effective automated log analysis methods are well sought after. To enable automatic log analysis, the very first step is **log parsing, Category G**, followed by applications of automated log analysis, *i.e.*, **anomaly detection, Category H**, **runtime behavior, Category I**, and **performance, failure, and fault diagnosis, Category J**, that we review in the following.

### Category G: Log Parsing

Log parsing is the process of converting the free-form text format of log files to structured events. Figure 2.15 provides an example of a raw log message from a log file that is parsed to its individual elements. Each log message is printed by a logging statement in the source code, which records a specific system event. Then, a log parser applies techniques

081109 203521 147 INFO dfs.DataNode$DataXceiver: Receiving block blk_-1608999687919862906 src: /10.250.14.224:35754 dest: /10.250.14.224:50010

| Parsed log | |
|---|---|
| Timestamp (date/hour/ms) | 081109/203521/147 |
| Log verbosity level | INFO |
| Source | dfs.DataNode$DataXceiver |
| Log template | Receiving block <*> src: <*> dest:<*> |
| Arguments | [“blk_-1608999687919862906”, “/10.250.14.224:35754”, “/10.250.14.224:50010”] |

Figure 2.15: Log parsing for a raw log message to a parsed log from HDFS logs [311].

to convert the free-form text format of the log messages to a structured format, as presented in Figure 2.15. More specifically, the log parser can extract useful information, such as timestamp, log verbosity level, variable arguments, and log template. The goal of the log parser is to convert each log message to a specific log template (*e.g.*, Received block $\langle * \rangle$ src: $\langle * \rangle$ dest: $\langle * \rangle$ in Figure 2.15). Ideally, there is a corresponding logging statement in the source code for each extracted log template, *e.g., log.info(“Received block %s src: %s dest: %s”, obj.blk_id, obj.src, obj.dest)*. The better the log parser can match the log templates with actual log printing statements in the source code, the merrier the quality of log parsing, and consequently, the more accurate log analysis tasks that follow.

Log parsing, traditionally, started with defining manual regular expressions to extract log templates and arguments. However, this approach alone is no longer efficient due to the huge number of log templates as well as their continuous evolution [347]. For example, Xu *et al.* [311] explained that in a Google’s system, on average, thousands of new logging statements are added every month. Therefore, automating the log parsing process is very well obliged. Some studies have also proposed, as an alternative, the use of static methods to curate the software’s source code and extract log patterns directly from the logging statements within the source code [311, 233]. These approaches are only useful if the source code of the system is available. To extend the application of log parsing to the scenarios that the source code is not available, *e.g.*, proprietary software, other studies have proposed various data mining approaches to extract log templates from the log files instead, such as frequent pattern mining in SLCT [299] and its extension, LogCluster [300],

| Reference - Aim | Experiments | Results | Pro | Con |
|---|---|---|---|---|
| Qiu [258] *et al.* - Designs *SyslogDigest* that extracts log events from the router's syslogs. | Syslog data from two large operational networks: a tier-1 ISP backbone network and a nationwide commercial IPTV backbone network. | It combines an *offline* and *online* domain knowledge learning components automatically extracts relevant domain knowledge from raw syslog data. The authors showcase the applications in network *troubleshooting*, and *health monitoring and visualization*. | Applies associated rule mining, then transforms and compresses low-level raw syslog messages into their prioritized high-level events. | The work is limited only to event template extraction from a specific system, *i.e.*, syslog messages. |
| Taerat *et al.* [292] - Introduces *Baler*, a token-based log parsing tool. | Logs of four supercomputers: *BG/L*, *Liberty*, *Spirit*, and *Tbird*. | For clustering, Baler relies on token attributes rather than frequency or entropy of token positions that are applied in other log parsers. Baler handles large datasets better than compared tools and more efficiently, *i.e.*, faster execution time. | Requires only one pass to cluster log messages based on their event templates. | Baler relies on the user to provide a dictionary of words. |
| Tang *et al.* [294] - Proposes *LogSig*, message signature based algorithm to generate events from textual log messages. | Logs of five real-world systems, including *FileZilla*, *PVFS2*, and *Hadoop*. | It searches for the most frequent message signatures and then categorizes them into a set of event types. LogSig performs better in *the quality of the generated log events* (F-Measure) and *scalability* when compared to prior work. | LogSig converts each log line into a set of ordered token pairs and then partitions log messages into $k$ different groups based on the extracted term pairs. | LogSig has a prolonged execution time on large log datasets and reaches low accuracy on the BGL data [146]. |
| Vaarandi *et al.* [300] - Presents *LogCluster*, a data clustering and pattern mining algorithm for textual log lines. | A set of six system logs from a large national institution, including database systems, mail servers, and firewall logs. | LogCluster improves SLCT [299] such that each Cluster $C_i$ is uniquely identified by its pattern $P_i$, and each pattern consists of words and wildcards, which makes it insensitive to word shifts. LogCluter performs more accurate clustering and finds fewer groups compared to SLCT. | The *support* of a cluster is calculated as the number of elements in that cluster: $supp(p_i) = supp(C_i) = \|C_i\|$. Finally, clusters with support of equal or higher than a threshold value, $s$, are selected. | The algorithm requires a two-pass process to categorize the list of frequent patterns. |

Table 2.12: Log parsing research - Topic (G).

| Reference - Aim | Experiments | Results | Pro | Con |
|---|---|---|---|---|
| Du *et al.* [106] - Proposes *Spell*, an online log parsing method based on longest common subsequence (LCS). | Two supercomputer logs: *Los Alamos HPC log* and *BlueGene/L log*. | Parses unstructured log messages into structured events types and parameters in an unsupervised streaming fashion with linear time complexity. Spell with pre-filtering has a faster computation time and achieves a higher accuracy compared to prior work. | The LCS approach achieves a faster template searching process by enabling subsequent matching and prefix trees. | The prefix tree depth can grow arbitrarily without limitation, which can lead to lengthy computation time on large datasets. |
| He *et al.* [148] - Proposes *Drain*, a fixed-depth tree-based online log parsing method. | Five real-world data sets: *BGL*, *HPC*, *HDFS*, *Zookeeper*, and *Proxifier*. | Constructs a tree data structure and groups the logs that belong to similar log events (*i.e.*, templates) into the same *leaf node* of the tree. The approach achieves higher or equal accuracy and obtains 51.85% to 81.47% faster runtime compared to other online log parsers. | Drain is specifically useful for web services as it enables log parsing in a streaming manner, and evaluation shows that Drain outperforms prior offline and online log parsing approaches. | It appears that Drain does not fully handle positional shifts in the log templates, and log messages that belong to the same log event but have different lengths will be grouped separately. |
| Messaoudi *et al.* [225] - Introduces *MoLFI* (Multi-objective Log message Format Identification), which leverages an evolutionary algorithm for log message format identification. | Six real-world datasets: *HDFS*, *BGL*, *HPC*, *Zookeeper*, *Proxifier*, and one industrial software logs. | MoLFI achieves a higher performance than the compared alternative algorithms in detecting the correct log message templates. | Formulates the log template identification task as a *multi-objective optimization problem* and propose a search-based solution based on the NSGA-II algorithm [100], *i.e.*, a sorting genetic algorithm. | MolFI suffers from low efficiency (i.e., high execution time) on large datasets as its iterative genetic algorithm, NSGA-II, is computationally intensive [347, 110]. |
| Dia [96] - Introduces *Logram*, which uses *n-grams dictionaries* to perform log parsing. Logram initially calculates the number of appearance of each n-gram (*i.e.*, token) in the log file. | Eventuated on 16 publicly available logs including *Android, BGL, HDFS, Spark, and Zookeeper* logs. | Achieves a higher parsing accuracy than the prior work, and it is 1.8X to 5.1X faster than in its calculation when compared to prior work. | Based on the threshold of an n-gram occurrence, Logram decides dynamic and static parts of the log messages and extracts the log templates. | One caveat for this log parser is the threshold selection for n-gram appearance, and if a dynamic n-gram occurs frequently, it might be mistakenly picked as a part of the log template. |

Table 2.12: Log parsing research - Topic (G) (continued).

iterative partitioning in IPLoM [217], clustering in LKE [116], longest common subsequence in Spell [106], search-space multi-objective optimization approach in MoLFI [225], parsing trees in Drain [148], and n-gram dictionary-based in Logram [96]. Contrary to the regular-expression-based and static analysis methods, these techniques are capable of extracting log templates from log files without needing access to the source code. After the logs are parsed, they are used for various applications, such as anomaly detection and failure diagnosis. Zhu et al. [347] presented a log parsing benchmark available here [38], and El-Masri *et al.* [110] performed a survey of log abstraction techniques. Table 2.12 summarizes log parsing research and Figure 2.14 categorizes various trends and goals for log file analysis after log parsing is applied.

### Category H: Anomaly Detection

Execution logs are extensively leveraged to monitor the health of software systems, identify abnormal situations, and detect anomalies that can lead to system failures. Anomaly detection methods include various approaches, such as: ❶ creating a state machine of normal execution and comparing the failure runs with normal models [167, 116], ❷ PCA-based approach which projects event logs to normal and abnormal subspaces [311], ❸ deep learning approaches which learn an LSTM model from normal execution workflows [107], and unstable logs [337], ❹ semi-supervised deep learning approaches with probabilistic label estimation [315], ❺ a statistical approach using probabilistic suffix trees [60], and ❻ cloud deployment by correlating logs and resource metrics [112]. As a reference for further reading, He *et al.* [150] performed a quantitative comparison of various log-based anomaly detection approaches. Table 2.13 provides a detailed comparison and *pros* and *cons* of various research for anomaly detention with logs.

### Category I: System's Runtime Behavior

Researchers have also utilized logs for monitoring the system's runtime behavior. Some of the research overlaps with approaches for *'anomaly detection'* and *'performance and failure diagnosis'*. These approaches include: ❶ using logs to customize operational profiles for industry software [140], ❷ web service composition [296], ❸ detecting inter-component interaction [242], ❹ mining system events correlation [118, 102], ❺ assisting developers in cloud deployments [278], ❻ performance model derivation [57], ❼ big-data analytics for cloud deployment [278], and ❽ detecting impactful system problems [149]. We provide further comparison for this research category in Table 2.14.

| Reference - Aim | Experiments | Results | Pro | Con |
|---|---|---|---|---|
| Xu *et al.* [311] - Applies the Principal Component Analysis (PCA) method to find unusual patterns in logs and identifies log segments that are likely to indicate runtime anomalies and system problems. | Logs of the *Darkstar online game server* and *HDFS*. | PCA extracts $k$ principal components by finding the axes with the highest variance among high-dimensional data. The approach can detect anomalous logs with high accuracy and few false positives while being efficient in its computation time. | For anomaly detection with PCA, two subspaces, *i.e.*, normal, $S_n$, and abnormal, $S_a$, are created. $S_n$ is created with the first k principal components, and $S_a$ is constructed with the remaining components. | Relies heavily on log parsing step to extract log structure from the logs and detect event sequences, and will fail if log messages do not follow the preferred structure. |
| Fu *et al.* [116] - Introduces a method to detect anomalies by converting unstructured log files to log keys. | Two distributed systems: *Hadoop* and *SILK*. | The research learns a finite state automaton (FSA) from the training set log keys to model the normal behavior of the system. The results show that the approach can detect system issues, such as workflow errors. | With the FSA and a performance model, the authors can identify anomalies in newly generated log files. The work detects three types of anomalies: 1) *work flow error*, 2) *transition time low performance*, and 3) *loop low performance*. | The approach does not work properly for *loop low performance* detection, results in false positives, and it is workload dependent. |
| Lou *et al.* [212] - Proposes an approach to detect anomalies by mining program invariants (IM) that have a clear physical manifestation. | Experiments on *Hadoop* and *Microsoft CloudDB*. | It detects anomalies if the new logs break certain invariants, *e.g.*, if an "open file" log message appears without observing a "close file", this invariant is violated and an anomaly is detected. Generally, produces fewer false-positive cases compared to the PCA-based approach. | Improved upon the PCA-based method [311], this approach provides the operators with intuitive insight (*i.e.*, what invariant is breached) on anomalies, and, hence, facilitate faster anomaly track down. | This approach is not able to detect anomalies that no invariant is broken, *e.g.*, many files are opened and closed continuously. |
| Chuah *et al.* [87] - Presents *ANCOR* that connects resource usage anomalies with system problems with logs. | *Ranger* supercomputer logs in two formats: *syslogs* and *ratlogs* (rationalized logs) [138]. | Evaluates the effectiveness of three different algorithms, *PCA*, *ICA*, and *Mahalanobis distance*. The results reveal a list of events with a strong correlation with system problems, such as *soft lockup*. | Performs anomaly and correlation analyses to detect the cluster nodes and jobs that are associated with the extra system resource usage that lead to system failures. | The approach cannot detect system problems that are not manifested as extra resource usage. |
| Du *et al.* [107] - Presents *DeepLog*, an online LSTM-based approach, to model system log files as natural language sequences. | *HDSF* and *OpenStack* log datasets. | DeepLog decodes the log message, including timestamp, log key, and parameter values, and applies both *deep learning* and *density clustering* approaches. The approach outperforms PCA and IM methods and produces a lower number of false-positive and false-negative cases. | Learns log patterns from the normal execution and constructs workflows, and detects anomalies when running log patterns deviate from the normal execution. | DeepLog is evaluated on systems with highly regulated logs and with a limited log key space, *i.e.*, Hadoop (29 keys) and OpenStack (40 key) [282]. |

Table 2.13: Log anomaly detection research - Topic (H).

| Reference - Aim | Experiments | Results | Pro | Con |
|---|---|---|---|---|
| Bertero *et al.* [65] - Leverages natural language processing techniques for anomaly detection. | 660 syslog log files, half of them (330 files) for normal system executions, and the other half are abnormal runs. | Explores the performance of three learning classifiers, *i.e.*, *Naive Bayes*, *Random Forest* (RF), and *Neural Networks*, and evaluates their performance on predicting normal versus stressed (*i.e.*, abnormal) log files. RF has the best performance. | Applies a word embedding technique, *i.e.*, *word2vec*, to map log message words to metric space, and then utilizes machine learning classifiers to summarize log files to single points. | The approach is limited to supervised learning and requires the pre-labeling of log files. |
| Bao *et al.* [60] - Utilizes both the source code analysis and the log file mining for anomaly detection. | *CloudStack* and *HDFS* logs. | It presents a probabilistic suffix tree-based statistical approach to detect anomalies from console logs. The results show the proposed approach can detect the largest number of anomalies compared to prior work. | The source code analysis employs control flow and log statement analysis to extract *"schema"* for the subsequent log parsing stage. | For feature extraction, the approach only takes into consideration the number of occurrences of an event and does not consider the sequential relationship of the traces. |
| Farshchi *et al.* [111, 112] - Proposes a regression-based statistical approach to correlate operation behavior with cloud metrics. | Experiments on *Amazon Web Services* (AWS) logs. | For anomaly evaluation, the work injects faults in 22 iterations of a rolling upgrade task and utilizes the learned model for fault prediction. Two-minute-time-window (2 mTW) metric observation prior to the anomaly achieves the highest F-Measure. | The authors utilize a regression-based method to detect the most statistically significant metrics for anomaly detection and observe cloud metrics changes and signal anomalies in case of divergence. | The evaluation is performed with synthetically injected faults. Further evaluation on actual system faults can help to better validate the approach. |
| Meng *et al.* [224] - Proposes *LogAnomaly*, an approach to model log messages as natural language sequences for anomaly detection. | Two datasets: *BGL* and *HDFS*. | LogAnomaly achieves a better F1-Score (0.96 on BGL and 0.95 on HDFS) compared to DeepLog (0.90 on BGL and 0.88 on HDFS). | It leverages a new word embedding approach, *template2Vec*, to model the sequential and quantitative patterns of logs and extract the semantic information of log templates. | The approach does not take into account the runtime parameter values. |
| Zhang *et al.* [337] - Proposes a log-based anomaly detection approach, *LogRobust*, which can handle unstable log lines. | Real and synthetic *HDFS* log data, and Service X logs from Microsoft. | It extracts semantic information of log events as semantic vectors. LogRobust achieves the highest F-Measure in detecting anomalous log lines in unstable datasets, and the performance decreases as the unstable logs change further (*i.e.*, become more unstable). | It can identify new but semantically similar log events that emerge from logging evolution and processing noise. | If there are drastic and significant changes to the entire code base or logging mechanism, the LogRobust would perform poorly in anomaly detection. |

Table 2.13: Log anomaly detection research - Topic (H).

| Reference - Aim | Experiments | Results | Pro | Con |
|---|---|---|---|---|
| Zhang *et al.* [332] - Proposes Anomaly Detection by workflow Relations (ADR). | *BGL* and *HDFS* logs. | The approach mines numerical relations from logs and uses the relations for anomaly detection. ADR detects a higher number of relations in less time compared to the invariant mining (IM) approach. | For faster online anomaly detection, ADR leverages an optimization approach, Particle Swarm Optimization (PSO) [173], to find the proper window size to split the log entries. | The experimentation is performed on highly regulated logs with a low number of log keys (*i.e.*, Hadoop and BGL), and for BGL data, simpler approaches, *e.g.*, SVM, outperform ADR. |
| Huang *et al.* [160] - Proposes *HitAnomaly*, that is a transformer-based [301] log anomaly detection method. | Three system logs: *HDFS*, *BGL*, and *OpenStack*. | The approach achieves F1-Scores higher than prior work for stable logs, and for unstable logs performs best under 10% instability injection into the log lines. | The approach allows to capture the semantic information in both log template sequence and parameter values and provides an attention-based classifier for log anomaly detection. | HitAnomaly's performance drops lower than LogRubost [337] for higher rates of instability in log sequences. |
| Zhou *et al.* [343] - Proposes *LogSayer*, a log-based anomaly detection approach with pattern extraction for cloud environment. | One *HDFS* log set and two *OpenStack* log data sets. | The key observation is that different components of cloud systems show different levels of system resource usage during anomalous behavior. The approach performs the best in detecting transient anomalies with an accuracy of 93% and outperforms DeepLog [107] and CloudSeer [321]. | Applies a back-propagation (BP) LSTM-based approach to learn and correlate the historical logs with current logs, and deviations are signaled as potential anomalies. | LogSayer's performance is dependent on the *time window* size, and its performance towards unstable logs [337] is not evaluated. |
| Chen *et al.* [84] - Proposes *LogTransfer*, to transfer anomalous log knowledge from the source system to the target system. | Proprietary switch logs over a two-year period, and *Hadoop* application and *HDFS* logs. | LogTransfer still requires anomalous instances of the target system for optimal performance. It achieves 0.84 *F1-score*, better results than unsupervised and supervised approaches, such as DeepLog and LogAnomaly. | It applies *GloVe* [252], an unsupervised word representation technique, to convert words in log templates to fixed-dimension vectors. | In the comparison section, unsupervised methods, *e.g.*, DeepLog is supposed to be trained on normal logs and not on a mix of normal and abnormal logs [107]. |
| Yang *et al.* 2021 [315] - Introduces *PLELog*, a semi-supervised anomaly detection through execution logs. | Experimented on *BGL* and *HDFS* logs. | With probabilistic label estimation (PLE), it can automatically assign labels to unlabeled datasets. PLELog outperforms compared semi-supervised and unsupervised anomaly detection approaches in terms of F-Measure. | It leverages an attention-based GRU neural network to detect anomalies. | The effectiveness of PLELog falls short compared to some prior anomaly detection approaches, *e.g.*, LogRobust [337]. |

Table 2.13: Log anomaly detection research - Topic (H) (continued).

| Reference - Aim | Experiments | Results | Pro | Con |
|---|---|---|---|---|
| Tang *et al.* [296] - Proposes a log-based approach to identify *service composition patterns* by finding associated services using *Apriori algorithm*. | A case study on 74 service-oriented applications. | The approach can detect service composition patterns from control flow with a high accuracy. | The approach first starts with collecting and preprocessing of execution logs, and continues with identification of frequent web services, and then extrapolates the control flows. | The approach fails to extract service patterns in cases that control flow has alternative branches. |
| Oliner and Aiken [242] - Proposes an approach to infer the interactions among the components of large-scale systems by analyzing logs. | Log of eight systems: four supercomputers (*Blue Gene/L*, *Thunderbird*, *Spirit*, and *Liberty*), two data clusters (*Mail Cluster* and *Junior*), and two autonomous vehicles (*Stanley* and *SQL Cluster*). | Log data signal compression allows for the scalability of *'lag'* correlation, and with minimal loss, this approach identifies system's behavioral model. | Performs a two-stage analysis: 1) PCA compression to summarize the anomaly signals, and 2) lag correlation to identify if the signals relate to each other with a time lag. | The extracted signals show correlation and not a causal relationship, and in addition, manual analysis of a system administrator is required to make sense of the data. |
| Fu *et al.* [118] - Proposes *LogMaster*, a tool to mine correlations of events in log files of large-scale cloud and high-performance computing (HPC) systems. | Experimented on three system logs, namely: *Hadoop*, *HPC* cluster and *BlueGene/L*. | Results show the approach is successful in correlating events related to failures with acceptable *precision* scores but with lower *recall* rates. | LogMaster parses the log lines into event sequences where each event creates an informative nine-tuple, and then uses an algorithm, named Apriori-LIS, to mine event rules from logs, and measures the events correlations. | Experiments on cloud and HPC systems shows LogMaster can predict failures with high *Precision*, however, the *Recall* scores are low and require improvement. |
| Shang *et al.* [278] - Suggests using execution logs from the cloud environment to assist developers of Big Data Analytics (BDA) applications. | A case study on three Hadoop-based apps: *WordCount*, *PageRank*, and *JACK* (industrial). | The approach reduces the verification effort and reaches comparable precision with traditional keyword search methods in verifying cloud deployment procedures. | This approach exposes the differences between pseudo and large-scale cloud deployments and it points the developers' to examine the inconsistencies, and therefore, facilitates the deployment verification effort. | The approach suffers from a high number of false positives in flagging presumptive problematic log sequences, and results in low precision. |
| Awad and Menascé [57] - Proposes an approach to use system logs and configuration files to automatically extract performance models of the system. | Experiments on *Apache Tomcat access logs* from a multi-tier server. | The results show the method is effective in extracting the workloads and system model by parsing the system configuration files and log files. | The approach extracts the interaction patterns between servers and devices and the probability associated with each interaction. | The work assumes the log templates are known for the systems under analysis. |
| Di *et al.* [103, 102] - Proposes *LogAider*, an analysis tool that mines potential correlations between various system events for the diagnosis purpose. | Logs of *BlueGene/Q Mira* supercomputer [8]. | The approach shows effective correlation between fatal system events and job events, with both high precision and recall values (99.9-100%). | LogAider can reveal three types of potential correlations between log events: *across-field*, *spatial*, and *temporal*. | It uses a threshold to find correlation candidates, and the evaluation scores, such as *Precision* and *Recall*, appear to be threshold dependent and vary significantly with the threshold. |
| He et al. *et al.* [149] - Proposes *Log3C*, a clustering-based approach to detect system problems. | Three datasets of an online large-scale service system X from Microsoft (confidential). | Log3C achieves F1-measures values 0.91, 0.86, and 0.868 for three datasets, and outperforms PCA and Invariant Mining approaches. | It applies cascading clustering to cluster and match the log sequences efficiently, and then correlates the log sequence clusters with KPIs to identify the impactul problems. | The research only considers a single KPI, *i.e.*, *failure rate*, to correlate with the log sequences. The research can be enriched with the inclusion of additional KPIs. |

Table 2.14: System's runtime behavior research - Topic (I).

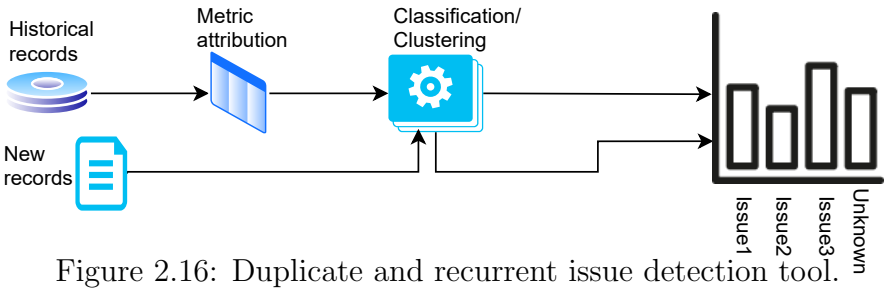


Figure 2.16: Duplicate and recurrent issue detection tool.

### Category J: Performance and Failure Diagnosis

In many cases, log messages are one of the most important clues and often the only available resource for the system's failure diagnosis and performance degradation, as it might be difficult and undeterministic to reproduce a failed scenario by replaying (*i.e.*, rerunning). Developers often have to diagnose a production run performance degradation or failure based on logs collected in the field and returned by customers without having access to the infield user's inputs. Performance and failure diagnosis approaches include: ❶ probable program execution paths investigation with logs [323], ❷ machine learning to compare and classify logs with good and bad performance [234], ❸ correlating performance counters (*e.g.*, CPU/memory usage) and logs [291], ❹ automaton-based workflow modeling [321], ❺ process-oriented dependability analysis [314], ❻ control and data flow analyses to extrapolate causal relations among longs [341, 340, 338], and ❼ fault diagnosis with logs [349]. Table 2.15 summarizes the research on mining of log files for performance and failure diagnosis.

### Category K: User, Business, Security, and Code Coverage

Other applications of log file mining include: **analyzing user statistics and behavior** [187], **application security** [244], **duplicate issue identification** [205], **code coverage with logging statements** [82], and **business analytics** [62]. For example, Figure 2.16 shows how recurrent issues are first classified through historical log records, and once a new record is available, it is analyzed and compared against the historical issues [205]. Table 2.16 further explains each research effort.

### Implementation and Evaluation

It is worth mentioning that many of the concepts with regards to feature selection, machine learning implementation, and evaluation metrics that are utilized for designing and

| Reference - Aim | Experiments | Results | Pro | Con |
|---|---|---|---|---|
| Cinque *et al.* [90] - Proposes a software fault injection approach to assess the effectiveness of logs in the recording of software faults in the deployed environment. | Three open-source systems: *Apache Web Server*, *TAO Open Data Distribution System*, and *MySQL Database Management System*. | Approximately, only 40% of the injected faults are covered by the existing logging statements in the three studies systems. | Faults are intentionally injected into the experimented software systems to determine the most common failure sequences and identify logging deficiencies and improve them. | The faults are synthetic (might not necessarily match real faults) and it requires access to the source code of the software. |
| Chuah *et al.* [88] - Presents an approach to reconstruct event order and establish correlations among log events to discover the root causes of a given failure. | *Syslogs* of *Ranger* and *Turing* supercomputers, and *BlueGene/L RAS* logs. | The authors received positive feedback from system admins that they have found the tool analysis useful in facilitating their diagnosing efforts. | Introduces a *Fault Diagnostics* tool *FDiag*, to discover faults, which comprises three components: a *Message Template Extractor* (MTE), a *Statistical Event Correlator* (SEC), and an *Episode Constructor*. | The approach depends on the availability of event-specific keywords as domain knowledge for correlation, and does not provide causality. |
| Yuan *et al.* [323] - Proposes *SherLog*, a tool that leverages runtime log information and source code analysis to infer the probable execution paths during a failed production run. | Evaluated on eight real-world software failures collected from different application such as *rmdir*, *Squid*, and *ln*. | The experiments show the information inferred by SherLog is useful to assist developers in failure diagnosis. | By accepting the execution log of a failed run and the source code, SherLog aims to identify what must or may have happened along the execution path. | SherLog relies on the amount of information available in log messages to perform its analysis. As such, log messages that lack the necessary debugging information will significantly limit Sherlog's effectiveness. |
| Pecchia *et al.* [251] - Conducts an experimental study to examine factors from event logs that help with the detection of failures. | Performs experiments on a set of 17,387 instances of injecting faults into three systems: *Apache Web Server*, *TAO Open DDS*, and *MySQL DBMS*. | Features such as system architecture, placement of the logging statements, and support provided by the execution environment can have an impact on the accuracy and effectiveness of the logs at runtime. | This research additionally investigates the logging improvement that can potentially increase the usefulness of the execution logs. | The approach requires access to the source code and is only tested on open-source projects. |
| Nagaraj *et al.* [234] - Presents *DISTALYZER*, a tool which utilizes log data to assist developers in diagnosing performance problems. | Case studies on three systems: *TritonSort*, *HBase*, and *BitTorrent*. | Results show that *DISTALYZER* is able to uncover undiagnosed performance issues for the experimented systems. | DISTALYZER uses machine learning techniques (*i.e.*, *Welch's t-test [306] and dependency networks [154]*) to compare log files with acceptable and unacceptable performance. | DISTALYZER leverages *ad-hoc* approaches (*e.g.*, thread id) to group log messages, which limits its application for less-structured logs. |

Table 2.15: Performance, fault, and failure diagnosis research - Topic (J).

| Reference - Aim | Experiments | Results | Pro | Con |
|---|---|---|---|---|
| Fronza *et al.* [115] - Proposes an approach to perform log-based prediction by applying *Random Indexing (RI)* and *Support Vector Machines (SVMs)*. | Experimented on log files of a large European manufacturing company (anonymous). | According to the findings, weighted SVMs achieve the best performance by slightly shrinking specificity (true negative rate) scores to improve sensitivity or recall, and specificity stays greater than 0.8 in the majority of the experimented applications. | It applies weighted SVM, which utilizes cost-sensitive learning to achieve balanced TPR and TNR values, and makes the method more reliable in classifying both failures and non-failures. | SVM classification performs well in classifying non-failure instances, but poor in identifying failures, *i.e.*, low *true positive rate* or *recall*. |
| Syer *et al.* [291] - Proposes an approach that combines performance counters and execution logs to diagnose memory-related issues in load tests. | A case study of *WordCount* application on *Hadoop*. | The approach flags less than 0.1% of the execution logs with a high precision of (⩾80%). | After clustering the events, authors apply scoring techniques to identify clusters that are abnormal and can be associated with a performance issue. | The approach has limited applications to memory performance issues, such as memory leaks, spikes, and bloats. |
| Zhao *et al.* [341] - Proposes *lprof*, a *log profiling* tool that recreates the execution flow of distributed applications. | Evaluated on four distributed systems: *HDFS*, *Yarn*, *Cassandra*, and *HBase*. | lprof's reaches 88% accuracy in attributing log messages to requests, and 65% of the diagnostics are helpful for the operators. | lprof performs control-flow (CF) and data-flow (DF) analyses, and infers if log messages are causally related and what variables are unmodified between multiple log printing statements, and then groups the logs and use them for diagnosing performance issues. | The lprof's static analysis is limited to a single software component and needs to be readjusted for different languages (bytecode), which is cumbersome in practice. |
| Xu *et al.* [313, 314] - Utilizes system logs to provision rolling updates in a cloud environment for process oriented dependability (POD) analysis. | Experiments with rolling upgrade on AWS with injecting 8 different types of faults into the cloud-based clusters. Faults include *machine image (MI) change during upgrade*, *key pair management fault*, and *security group configuration fault*. | The evaluation results show acceptable performance (90+%) in precision, recall, and accuracy scores in diagnosing the injected sporadic faults. | It creates a process model of the desired provisioning activities through log data with added annotation and checkpoints. The deployment logs are checked and assertions are raised in case there has been a deployment violation. | The approach heavily relies on the specific information in the logs and the absence of this information severely impacts the performance of error detection. Following research [112] aims to combine logs with system metrics for a more robust analysis. |
| Russo *et al.* [269] - Proposes an approach to mine and learn error predictors from system logs, and then applies it to a real telemetry system for failure prediction. | Experimented on log sequences of 25 different applications of a software system for telemetry and performance of cars. | The evaluation achieves 78% *recall*, and 95% *precision*. | Uses three popular support vector machines (SVMs): multilayer perceptron, radial basis function, and linear kernels - to learn and predict *defective* (*i.e.*, faulty) log sequences. | The study is performed on the logs of a single system. Thus, the result of the applicability of the proposed approach to other systems and other software domains remains unknown. |

Table 2.15: Performance, fault, and failure diagnosis research - Topic (J).

| Reference - Aim | Experiments | Results | Pro | Con |
|---|---|---|---|---|
| Yu *et al.* [321] - Introduces *CloudSeer*, a lightweight and non-intrusive approach that works on interleaved logs for cloud workflow monitoring. | CloudSeer is prototyped and evaluated on an open-source platform, *i.e.*, multi-user *OpenStack* logs. | The approach is accurate enough to check and infer workflows for most interleaved log sequences. Cloudseer reaches 92+% accuracy in checking interleaved logs for six experimented groups with a satisfactory checking efficiency (*i.e.*, computation time). | CloudSeer constructs *automatons* for the workflow of management tasks based on their normal execution scenarios, and later checks log messages against these *automatons* to detect workflow discrepancies and divergences in an online approach. | The performance problems are only detected for log entries with *'ERROR'* log verbosity level, and model creation requires multiple executions of a single task. For messages which do not accompany an error, finding a timeout is not trivial and requires further discussion. |
| Gurumdimma *et al.* [135] - Introduces *CRUDE*, which combines console logs with *resource usage* data to improve the error detection accuracy in distributed systems. | Experimented on *Rationalized logs* (ratlogs) from the Ranger Supercomputer containing four weeks worth of data: resource usage data (32GB) and rationalized logs (1.2GB). | The approach is able to identify 80% of errors leading to failures, and achieves f-measure over 70%. | The approach has three main steps: it clusters nodes with similar behavior, then uses an anomaly detection algorithm to detect jobs with anomalous resource usage, and finally, links anomalous jobs with erroneous nodes. | The proposed approach does not model temporal relationships to improve fault identification [288]. |
| Zhao *et al.* [340] - Introduces *Stitch*, a distributed and end-to-end performance profiler by flow reconstruction. | Evaluated both through a controlled user study and lab experiments on *Hive*, *Spark*, and *OpenStack*. | On average, Stitch achieves 96% and 95% accuracy for object and edge detection, respectively, for workflow reconstruction. | Stitch aims to construct the system model and the hierarchical relationship of objects in a distributed software stack without requiring domain-specific knowledge. | Although Stitch can establish correlations between different software's objects and modules, it cannot accurately infer causal relationships among them. |
| Zou *et al.* [349] - Proposes *UiLog*, which is a fault analysis tool, to collect logs and their statistics from various components and diagnose the detected faults. | Performs experiments on logs of components (*e.g.*, disk, I/O, memory) of a cloud environment, *StrongCloud*, collected over a year period. | Twelve categories of faults are detected, and fault detection precision is maxing out at 88% when the length of logs is more than 200 words. | The approach classifies logs by the fault type in real-time and performs fault correlation analysis to help administrators and locate the faults' root causes. | Requires domain knowledge, and the precision of fault analysis is dependent on the size of the logs. |
| Zhang *et al.* [338] - Introduces *Pensieve*, a flow reconstruction tool for performance failure reproduction through system logs and bytecode. | Evaluated on 18 randomly sampled real failures on four systems: *HDFS*, *HBase*, *ZooKeeper*, and *Cassandra*. | Pensieve is able to reproduce 72% of the sampled failures within ten minutes of analysis time. | Pensieve leverages event chaining, and extrapolates a chain of causally dependent events leading to the failure while using *partial trace observation* technique, which significantly limits the execution paths to observe. | Some domain-specific knowledge or a developer familiar with the system is needed to actually describe and diagnose the failure, and make sense of the chain of the events. |

Table 2.15: Performance, fault, and failure diagnosis research - Topic (J) (continued).

| Reference - Aim | Experiments | Results | Pro | Con |
|---|---|---|---|---|
| Lee *et al.* [187] - Employs a unified logging infrastructure in Twitter to perform analysis on the user statistics by the use of log data. | *"Client events"* within Twitter logging framework. | Discusses a variety of applications for the proposed approach, such as *summary statistics*, *event counting*, *funnel analytics*, and *user modeling*. | The research applies techniques from natural language processing (NLP) to process the user's behavior on the website; the user's behavior right now is strongly influenced by immediately preceding actions. | Currently, user session sequences only capture event names and do not provide enough details for more sophisticated types of analyses. |
| Lim *et al.* [205] - Addresses the problem of automated identification of recurrent and unknown performance issues. | Evaluated on two datasets: Transaction Processing Performance Council BenchmarkTM W (TPC-W) [19], and System X, a real production system (confidential). | The results show the approach achieves higher *AUC* when compared to approaches such as *Fingerprint*, *Signature*, *K-means*, and *Hierarchical* approaches in recurrent and unknown issues identification. | The approach works based on the mining and metric extraction of historical log records, and utilizes a ***H**idden **M**arkov **R**andom **F**ield (HMRF)* based approach for the clustering of recurrent issues. | The proposed statistical analysis only works if a large amount of monitoring data over a long period of time is available. |
| Oprea *et al.* [244] - Proposes a framework that analyzes log data collected at the enterprise network borders on a regular basis (*e.g.*, daily). | Experimented with DNS logs released by *Los Alamos National Lab (LANL)*, and *AC* dataset of web proxies logs generated at the border of a large enterprise network (confidential). | The approach can detect malicious web domains with high accuracy and low false-negative rates. The work also detects new malicious domains that are not previously reported/detected by other tools in the enterprise. | Creates a bipartite graph $G = (V, E)$, such that hosts and domains are vertices on each side of the graph. There will be an edge between a host and a domain if the host connects with the domain. | The approach cannot detect regular connections to malicious domains which happen in the training phase. |
| Barik *et al.* [62] - Performs a case study of log utilization in Microsoft for business decisions and analytics. | Performed interviews with 28 engineers at Microsoft, and followed that up with a survey of 1,823 respondents to confirm their findings. | Use of log event data is pervasive within the organization and the usage primarily falls into eight categories, among them *engineering the data pipeline*, *instrumenting for event data*, *troubleshooting problems*, and *making business decisions*. | This research highlights that event log data surely plays an important role in the company's decision-making process as the industry makes a transition towards a data-driven decision-making paradigm. | The study is performed on a single software company and the findings may not generalize to other software and institutions. |
| Chen *et al.* [82] - Presents an approach, called LogCoCo (*i.e.*, **Log**-based **Co**de **Co**verage), to estimate and measure the source code coverage using the readily-available execution logs. | Five commercial proprietary projects from *Baidu* and one open-source project, *i.e.*, *HBase*. | Measures the accuracy and usefulness of LogCoCo, and it achieves high accuracy for different types of code coverage (*Must* and *May* have been executed). Additionally, the tool's results are useful to evaluate and improve the test suites for code coverage. | Using program analysis techniques, LogCoCo matches the execution logs with their corresponding code paths and estimates three different code coverage criteria: 1) *method coverage*, 2) *statement coverage,* and 3) *branch coverage*. | The approach cannot accurately infer whether a *May* executed code region is actually covered in a test. |

Table 2.16: User, business, security, and code coverage research - Topic (K).

assessing the log statement automation approaches (Sections 2.5.3-2.5.3) are also leveraged for designing and evaluating log file mining tasks. For example, *DeepLog* [107], proposes a machine learning algorithm for log file mining. LogRobust [337] extracts feature vectors from the log files and implements an LSTM deep learning approach, and evaluates its anomaly detection approach with *Precision*, *Recall*, and *F-Measure*. *Drain* [148], a log parsing tool, evaluates its performance with *F-Measure*.

### 2.5.5 Category L: Emerging Applications of Logs

Thus far, we discussed different logging practices and log applications, mainly in large-scale software systems. However, most recently, there has been a special interest in applications of logs in other domains such as mobile devices [86, 330], embedded [105, 129, 304], and big data [228, 274, 222]. We summarize the key findings here:

- Prior studies have proposed the application of logs for emerging areas such as **mobile** and **big data systems**.
- For **mobile**, developers should be aware of different logging practices that might apply to alternative platforms with different design criteria and requirements. For example, because mobile devices operate on battery with limited storage space, the cost and overhead of logging (*e.g.*, continuously flushing logs) become more exorbitant and unfavorable than software that is running on a workstation.
- Developers of log analysis tools have considered **big-data platforms** to scale and speed up log analysis.
- **Natural language attributes** of logs [145, 126] open up a new avenue for log statement automation, *e.g.*, log statement description, and automated log analysis of logs, *e.g.*, anomaly detection.

Table 2.17 summarizes research in **Category L**.

**Finding 5.** *Researchers have contributed to logging research in various categories, and research continues to progress in the existing categories and also grows to the emerging domains. Prior research spans through* ***logging cost and practices****,* ***mining and automation of logging code****, and* ***mining and automated analysis of log files****.*

| Reference - Aim | Experiments | Results | Pro | Con |
|---|---|---|---|---|
| Miranskyy *et al.* [228] - Discusses the challenges of event log analysis for big data systems (BDS), as the logs generated by a BDS can be big data themselves. | Categorizes seven challenges of log analysis for big data systems. | Highlights currently available solutions to each challenge and discusses unanswered questions, based on the authors' and industrial experience. | The authors categorize the challenges of big data log processing into seven classes, including **scarce storage**, **unscalable log analysis**, **inaccurate capture & replay**, and **inadequate tools for instrumenting BDS source code**. | Accurate mapping and case studies and examples of challenges in real-world big-data software can further illustrate the current issues. |
| Salman *et al.* [274] - Proposes *PhelkStat*, a tool for analysis of system event logs of large-scale data centers on Apache Spark's big data platform. | A set of public (*e.g.*, *Spirit*, *Thunderbird*, and *Liberty*) and private (*e.g.*, Cray and dartmouth/campus) log data. | Performs evaluation on a set of log analysis tasks such as *arrival rate distribution*, *anomaly content*, and *runtime analysis*. | The authors utilized a set of attributes, *i.e.*, temporal and spatial metrics such as arrival rate and byte count, to characterize system event logs and then correlate the metrics with the runtime performance of the system. | Log analysis tasks are partially correlated with system events, but are not analyzed meaningfully to draw actionable steps for admins and users of the system. |
| Mavridis *et al.* [222] - Evaluates various log file analysis tasks with two cloud computational frameworks, *Apache Hadoop* and *Apache Spark*. | Experiments on log files of an *Apache HTTP server*, and implemented useful log analysis tasks such as checking for denial of the service (DoS) attacks from the available logs. | Compared performance of Hadoop and Spark by evaluating their execution time, scalability, resource utilization, and cost and power consumption | The research showed the potential of utilizing distributed big-data platforms for facilitating log analysis. | The analysis needs to be expanded to designing tools that can leverage the parallelism of distributed big-data platforms to accomplish a faster and scalable analysis of logs. |
| Chowdhury *et al.* [86] - Performs an exploratory study to investigate the energy impact of logging in Android applications using GreenMiner [156], an automated energy test-bed for mobile applications. | Studies approximately a thousand versions of 24 Android applications (*e.g.*, CALCULATOR, FEEDEX, FIREFOX, and VLC) with logging enabled and disabled, accompanied by a controlled experiment on a synthetic application. | There is little to no energy impact when logging is enabled for **most** versions of the studied applications. However, about 79% (19/24) of the studied applications have at least one version that exhibits a noticeable impact on energy consumption. | The authors found that **the rate of logging**, **the size of messages**, and **the number of log buffer flushes** are significant factors of energy consumption attributable to logging on mobile devices. | More accurate models that correlate mobile log events with the amount of energy consumption are required. |

Table 2.17: Emerging log research - Topic (L).

| Reference - Aim | Experiments | Results | Pro | Con |
|---|---|---|---|---|
| He *et al.* [145] - Characterizes natural language attributes of log statements' descriptions. | Experiments on ten Java and seven C++ open-source projects and answers four research questions. | Findings confirm the natural characteristics of logs, such as endemic and specific. | Proposes an automated approach for log description prediction based on source code similarity and edit distance. | The dynamic part of the log statements is left out, and the approach for log description automation is limited to cases that a similar code snippet is found. |
| Zeng *et al.* [330] - Replicates the work of Yuan *et al.* [325] and investigated the logging practices in Android applications. | Performs a case study on 1,444 open-source Android applications in the F-Droid repository. | Although mobile app logging is less pervasive than server and desktop applications, logging is leveraged in almost all studied mobile apps, and there are noticeable differences between the logging practices applied in mobile applications versus the ones in server and desktop applications, as observed by prior studies [79, 325]. | The majority of the logging statements in mobile apps are in *debug* and *error* verbosity levels, while *info* level logging statements are the prevailing level in server and desktop applications. | The research can be expanded by providing developers with guidelines for mobile apps. |
| Gholamian and Ward [126] - Performs an experimental study on natural and local characteristics of log files. | Experiments on eight system logs (*e.g.*, *HDFS* and *Spark*), and two natural language language data (*e.g.*, *Gutenberg* and *Wiki*). | Six findings confirm that log messages are natural and local, even more or so than common English text. | Applies the findings and proposes an NLP-based anomaly detection approach from log files, which utilizes n-gram models. | More advanced NLP models (*e.g.*, deep learning and BERT) need to be investigated to improve the anomaly detection task. |

Table 2.17: Emerging log research - Topic (L) (continued).

## 2.6 RQ4: Challenges and Opportunities for Future Work

In the previous sections, we reviewed and discussed the state-of-the-art logging research by providing an introduction to log messages and log files (2.2), costs and benefits associated with logging (2.5.1), mining logging statements (2.5.2), automated logging approaches and their evaluation metrics (2.5.3), mining of log files (2.5.4), and finally, the emerging areas of log application (2.5.5). In this section, as we revisit each section, we put emphasis on answering RQ4 and specify future directions and opportunities for each category. We point out the missing pieces of the puzzle for each area, followed by our intuitive approaches for tackling those issues, which are inspired by the collective knowledge of the prior work. In the following, we include opportunities for future work based on the research categories.

### 2.6.1 Category A: Logging Cost

#### Adaptive and Constraint-Based Logging

An imperative trend that we foresee as future research is the need for adaptive logging [229]. On the one hand, continuously logging in details (*e.g.*, in trace verbosity level) can incur performance overhead, and on the other hand, logging very little might degrade the effectiveness of the logs. Therefore, we anticipate further research that will work on dynamically adjusting the amount of logged data from the least verbose to the most verbose level in order to help with detailed postmortem analysis, if the system is in a detected anomaly state, and on the other side, minimize the performance overhead of logging while the system operates normally and as expected.

#### Whether to Log?

We mentioned that the prior research has explored various challenges such as *'what to log?'*, *'where to log?'*, and *'how to log?'*. We see potential for further research on all challenges of logging and with more emphasis on **'whether or not to log?'**. With the emergence of adaptive logging and logging less when not needed and log more details whenever necessary, the idea of whether or not to ultimately print an existing logging statement becomes important. We foresee future research that explores different scenarios of whether logging statements are eventually printed or filtered based on the goal of the

logging analysis tasks, *i.e.*, performance evaluation or failure diagnosis, and the operating state of the system, *i.e.*, normal state vs. when a system anomaly is detected.

### 2.6.2 Categories B, C, D: Logging Practices, progression, and Issues

#### Improved Logging Practices

Although logging practices in the software development process have been reviewed and improved over the past decade [325, 78], there is still room for betterment [142]. Additional tools that can automatically detect log-related issues are required. Moreover, because the majority of current logging practices and decisions are *ad-hoc* and decided by developers on the spot, the introduction of systematic logging practices that can provide suggestions to developers while composing the code can ensure a higher quality of logs. Also, further research that can provide directives and insights for developers with regards to good versus poor logging practices, and hence help to improve their logging practices and make better use of logging, is of interest. For example, more effective logging can enable the customers of the software systems to solve problems themselves using the logs without relying on developers or avoid unnecessary logging costs, such as exposing users' sensitive information in the logs [192]. Another angle for logging practices improvement includes studies that investigate cost-aware logging, which can help developers to estimate and optimize the cost of logging while benefiting from the logs. Although efforts such as Log20 [339] have aimed to address this issue, there is still a sizable room to improve upon. This avenue of future research can be also expanded to other platforms such as mobile devices as the logging practices can be different depending on the applications and the system requirements [330]. Prior research has shown other areas such as mobile systems, that are not in the research community's spotlight, are even more in dire need of systematic guidance and automated support tool for assisting in logging practices [330].

#### Representation of Log Files

It is a safe assumption that the log analysis methods require, or at the very least, perform better on logs with good quality to conduct meaningful analyses. Therefore, we foresee future research in improving the formatting and defining universal structure for log messages, which will directly help in achieving more systematic organization of log files, and consequently, more effective log analysis with higher *Precision* and *Recall* values. This

goal is also partially realized with proper selection and improvement of logging libraries and utilities, *e.g.*, Log4j, SLF4J, and Logback.

### Logging Libraries and Utilities

Logging libraries and utilities (LLU) provide additional functionality, structure, and flexibility in logging for developers such as log verbosity levels and thread-safety [81]. Although LLUs facilitate logging, there has been insufficient research on this topic. Further research that aims to improve the performance of logging libraries by performing some of the logging tasks during the compile time is necessary [316]. Furthermore, the development of application-specific logging libraries will provide higher logging flexibility and better API for developers in a specific domain, similar to *Log++* [220] for cloud logging, to perform workload-related logging. For example, in a cloud deployment and provisioning process, users are further interested in logging the machine image initialization and termination steps in more details to enable better debugging in case of failures. Additionally, LLUs can improve to bring in new configurability, such as supporting different log verbosity levels for separate parts of a logging statement [192]. Another angle that LLUs can improve is to provide checks on the format of the developer's provided logging text and ensure that the provided content passes a minimum set of standards in order to make logs more useful and organized. Additionally, logging libraries can help to reduce the overhead of logging. The way that some of the LLU work is that, during the runtime, all of the logging statements are executed but based on the verbosity level of the logging statements, some of the logs are filtered from being written to the log file. This approach can still introduce a considerable overhead if logging statements make calls to other methods and variables. To cope with this situation, developers include log statement guarding (*e.g.*, putting the log statement inside an *if-clause*) to avoid the logging statement being executed based on whether or not the level is enabled. Therefore, this type of log guarding improvement would be beneficial to be implemented inside the LLUs [142].

### Application Specific Logging

As logging messages can provide valuable information with regards to the different aspects of the running software, it is also evident that different tasks and applications that rely on logs require different types of information from the log files. Therefore, we anticipate application-specific logging research in the future to grow. That is to say, depending on the application, we need to log different categories of runtime information. For example, for a security log, certain values need to be printed while not compromising sensitive

information; however, this might not be an issue in postmortem analysis of logs [264]. Additionally, developers of other platforms, such as mobile apps, should be aware of the differences between desktop/server and mobile practices as it comes to logging, as for mobile, there is energy overhead concern that should be taken into account [330]. Therefore, different platforms also might end up logging different information with varied frequencies of outputting logging statements.

#### Maintenance of the Logging Code

As the software systems continue to grow, maintaining the logging code becomes more challenging. Previous studies have observed that the logging code is not maintained as well as the feature code, as there is no straightforward way to test the correctness of the logging code [80]. Therefore, we emphasize that further research should consider the systematic maintenance and testing of logging code alongside the feature code evolution. Additionally, there are interesting opportunities for developing automated tools that can read the context of the feature code changes and suggest logging code maintenance and updates concerning the feature code updates while the new feature code is being checked in.

### 2.6.3 Category E: Log Printing Statement Automation

#### Automatic LPS Generation

In contrast to developer-inserted logs, LPS automation aims to auto-generate or suggest new logging statements or enhance the quality of currently available logs inside the source code based on various source code and application criteria. Although this topic has been of interest recently [326, 324, 346, 339, 166, 123, 203], considering the continuous advancement and birth of new AI and learning methods, we anticipate future research in the development of machine learning methods to implement and automate logging, with statistical modeling, supervised, unsupervised, and deep learning approaches will continue to foster. These methods should consider automating different aspects of the logging statements, such as the ***location***, ***content***, and ***verbosity level***. The automated methods can also consider different criteria for automation, such as diagnosability versus cost-awareness [192]. The ultimate goal is to achieve an automated approach that can introduce high-quality log suggestions or enhancements for various applications. Subsequently, assuming the development of different approaches, a comparative study of different approaches and the areas that each one performs better becomes necessary, similar to the comparison of different log parsing techniques in [347].

#### Constraint-based Logging

The majority of the log automation tools have aimed to mimic developers' logging habits [123, 166, 346]. In other words, the log learning approaches work to learn developers' logging habits to decide if a new unlogged code snippet requires a logging statement. However, prior work [142] has also shown that developers make mistakes, and in some places, they even forget to log in the first place. Thus, one remaining important challenge is to develop constraint-based automated logging approaches to guarantee a particular logging goal, *e.g.*, at minimum, one iteration out of 100 iterations of method *MtdM()* is logged, or a particular execution path is fully disambiguated with logging, *i.e.*, we can accurately determine which code segments *'must'* have been executed. Another example can be ensuring the beginning and the end of all methods of interest are logged. By doing so, we can guarantee that at least a minimum quality of logs is granted.

#### Golden Quality LPSs for Benchmarking

To the best of our knowledge, there is no prior work that quantitatively measures the quality of logging statements in each software project. Many of the prior work consider the developers' inserted logging statements as ground truth to evaluate their automated logging approach [346, 166, 123]. However, prior research has shown that there is no general guideline for logging and developers mostly rely on their intuitions and insert logging statements in an *ad-hoc* manner [209]. As such, defining a set of quantitative metrics that can be applied to evaluate the quality of logs on various software projects and give them scores can be highly beneficial. This allows to find projects with high-quality logs, learn from them, and use them as a *golden benchmark* for comparison with auto-generated logs.

### 2.6.4 Category F: Log Maintenance and Management

**Log collections and compressors.** Log collections, similar to LogHub [151], are useful for evaluating various log analyses. Additionally, datasets that are labeled and differentiate normal against abnormal log records are well sought after, as they enable the application of supervised and deep learning approaches for log analysis [337, 204, 147]. As such, we see value in further research to collect log data from various software and application domains, and develop automatic and accurate probabilistic methods [315] to label the data to facilitate log analysis and logging practices research. For log compression, because logs generally benefit from higher repetition than natural text, future research can benefit from designing and evaluating log-tuned compressors, which not only can result in more effective

compression but also more efficient and streamlined decompression, for later auditing and analysis.

### 2.6.5 Categories H, I, J, K: Automated Log Analysis Applications

#### Log Analysis and Tools

Prior research has proposed plenty of log analysis methods and tools for different applications, such as **anomaly and problem detection** [311, 116], **performance and failure diagnosis** [341, 340, 321, 338], **system's runtime behavior** [140, 149, 278], **system profile building** [244], **code quality assessment** [279], and **code coverage** [82]. Log analysis, starting with log parsing, plays an essential role in extracting useful information from the log files. As logs can be viewed in different ways, such as events, time series, and feature vectors, this enables different types of analyses. Complementary to the available research, because logs are non-intrusive and readily available, we anticipate new methods of log analysis or improvement of the current methods will be sought after for different applications. The quality of log analysis can directly impact the amount of actionable information that we can extract from the log files. Therefore, we expect new logging analysis approaches will emerge that utilize and combine a variety of algorithms to achieve a more accurate analysis. The approaches might also assume a specific format of logs, *e.g.*, log messages following a specific template within the log files, to achieve a more personalized analysis. For example, the research can benefit from considering multiple factors, such as the content of each log message, the frequency, and the sequencing of log messages in log analysis tasks, *e.g.*, anomaly detection, in order to achieve a deeper understanding of what happens in the logs. Lastly, we foresee that future research will benefit from utilizing AI approaches in understanding and leveraging the hidden semantics of the log messages, rather than solely focusing on learning log patterns and templates. This will enable a more sophisticated log analysis.

#### Scalable and Online Log Processing

In order to keep pace with the massive amount of growing logs in size (at the rate of approximately multiple terabytes per day [147]) and various formats, which is the by-product of the software growth as well as the number of software users' growth, we anticipate further research will be conducted to develop and update the current logging processing tools

and platforms. Thus, future research should consider leveraging distributed and parallel processing platforms (*e.g.*, Apache Spark) in conjunction with efficient machine learning approaches to implement scalable log analysis tools for all stages of the process, *i.e.*, real-time collection, processing, and storage of voluminous logs [73]. In addition, as many of the enterprise software platforms require 24/7 up-time and availability, the need for on-line tools that can perform the log analysis simultaneously as the system generates logs becomes more apparent. We require the tools to be efficient enough to perform analysis at the same speed or faster than the log generation rate.

### 2.6.6 Category L - Emerging Logging Research

#### Natural Language Processing of Logs

Prior work [155, 298] in software engineering has utilized natural language processing (NLP) for software tasks such as source code next token suggestion. Recently, there has been a thread of research on analyzing logging statements as natural language sequences. He *et al.* [145] characterized the NLP characteristics of LPS descriptions in Java and C# projects, and Gholamian and Ward [126] showed software execution logs are natural and local, and these features can be leveraged for automated log analysis, such as anomaly detection. We hypothesize that further research is required to confirm the NLP characteristics of software logs, and eventually, leveraging NLP characteristics of logs will further benefit automated log generating and analysis of log files. Moreover, the recent advancements in NLP models, *e.g.*, BERT models [101], calls for further investigation and application of them in improving the performance of log mining tasks. The intuition is that these models can embed and learn a higher degree of log semantics, and thus, can better enable actionable diagnosis from logs.

#### Log Summarization and Visualization

Prior works [259, 74, 210] have proposed approaches to summarize and visualize console and security logs. Log summarization and visualization is a natural response to the ever-growing scale of logs to gain high-level insight into the logs. In large-scale distributed software systems, as the scale of logs continue to grow, and various subsystems continue to generate logs in heterogeneous formats and rates, we foresee the development of approaches and solutions, both in academia and industry, that aim to make high-level sense of logs and to gain big-picture insight with visualizing logs. In addition, log summarization will help developers and practitioners to focus their troubleshooting efforts on a smaller set of

| No. | Avenue | Rationale | Selected research |
|---|---|---|---|
| 1. | **Adaptive logging** | Dynamic adjustment of the logging level from the least to the most verbose level helps with detailed postmortem analysis. | [229] |
| 2. | **Whether to log?** | Different scenarios of whether LPSs are printed or filtered based on the goal of the logging analysis tasks should be studied. | [104] |
| 3. | **Logging practices** | Future research can improve on logging practices in the software development process and reduce the *ad hoc* and forgetful developers' logging habits. | [325, 78, 142] |
| 4. | **Representation of the log files** | Further research can improve the formatting and standardization of log messages, which directly results in more organized log files and more accurate automated analysis. | [273, 240] |
| 5. | **Logging libraries and utilities** | Logging libraries and utilities (LLU) can provide additional functionality, structure, and flexibility in logging for developers. | [81, 316, 220] |
| 6. | **Application-specific logging** | Research can investigate and ensure that how different tasks and applications that rely on logs can record application-specific information (*i.e.*, based on the application needs) into the log files. | [264, 330] |
| 7. | **Maintenance of logging code** | As maintenance of the logging code becomes more challenging, future research requires to develop automated approaches to ensure up-to-date and issue-free logging code. | [80, 201, 78] |
| 8. | **Automated and constraint-based log generation** | Research requires to improve the quality of auto-generated LPSs and, also, enhance the quality of the developer-inserted *ad-hoc* logs by adding additional variables, *etc.* | [72, 203, 209, 200, 326, 324, 346, 339, 166, 123] |
| 9. | **Golden quality log statements** | High quality logs are required to learn from, and use them as a *golden benchmark* for comparison with auto-generated logs. | [209, 346, 166, 123] |
| 10. | **Log collections** | There is a need for labeled log data collections and development of automated log labeling approaches to facilitate automated log analysis. | [151, 40, 94] |
| 11. | **Log compression** | Compressors which are designed for logs are needed to improve the compression/decompression efficiency and enable efficient long-term storage and backup of logs. | [208, 319, 140] |
| 12. | **Log analysis for various objectives** | Research will actively continue to propose and improve approaches for more accurate log analysis for different log mining tasks and postmortem debugging. | [315, 311, 116, 341, 340, 321, 338] |
| 13. | **Scalable and online log processing** | Scalable and real-time log processing is required to keep pace with the massive amount of growing logs in size, *e.g.*, multiple terabytes per day. | [147] |
| 14. | **Natural language processing of logs** | Leveraging NLP characteristics of logs will benefit automated log generating (*e.g.*, log description generation) and NLP processing of log files. | [123, 155, 298, 145] |
| 15. | **Log summarization & visualization** | Development of approaches that aim to elicit and condense big-picture insights from logs with visualizing and summarization are in demand, and this will enable practitioner to only focus on significantly smaller but most important portion of logs. | [259, 74, 109] |

Table 2.18: Summary of avenues for future work in logging research.

relevant logs. Knowledge graph representation is a potential candidate for this aim [109]. Lastly, we provide a digest of the avenues for future of logging research in Table 2.18.

**Finding 6.** *Outstanding problems exist for each category of logging research. Future research can consider and tackle these challenges to improve the quality of log statements and log files, and thus enable more effective log analysis tasks.*

## 2.7 Conclusions

Logging statements and log files are the inevitable pieces of the puzzle in analyzing and ensuing various aspects of correct functionality of software systems, such as debugging, maintaining, and diagnosability. The valuable information gained from logs has motivated the research and development of a plethora of logging practices, logging applications, and log automation and analysis tools.

In this survey, we initially started with the basics of log statements and log files and the involving challenges in extracting useful information from them in Sections 2.1 and 2.2. As we conduct the survey, we aim to answer four crucial research questions related to software logging: **(RQ1)** categories of logging research, **(RQ2)** publication trends based on topics, years, and venues, **(RQ3)** available research in each category, and finally **(RQ4)** challenges and opportunities for future logging research. We next reviewed the costs and benefits associated with logging in Section 2.5.1 and followed that up with research that mines logging statements to derive logging practices in Section 2.5.2. In Section 2.5.3, we reviewed the proposed methods for automated logging, and we mentioned evaluation methods and metrics for auto-generated logs and learning-to-log platforms in Section 2.5.3. In section 2.5.4, we reviewed log file mining and log analysis research which aims to expedite and scale up the log processing, and apply logs for different system maintenance tasks such as anomaly and failure detection/diagnosis, performance issues, and code quality assessment. We also reviewed the emerging domains and applications for logging, such as in NLP, mobile, and big data in Section 2.5.5. Finally, we discussed the opportunities for future research in different aspects of logging statements and log files, their practices, and their analyses in Section 2.6.

Although current research advances have made logs more useful and effective, there are still multiple remaining challenges and avenues for future work and improvement. Categories of challenges remain in various aspects of **automated log analysis**, **LPS auto-generation**, **scalable logging analysis and infrastructure**, **cost-aware logging**, **log**

**maintenance and management**, and **improved logging practices**. We foresee future research in multiple directions for logging as follows:

- As the size of computer systems increases, we anticipate the voluminousness and heterogeneity of logs, which turns it into a big-data problem, will demand further quantitative cost analysis for collecting, processing, and storing of logs, as logging can infer computation, storage, and network overhead. Additionally, due to the voluminousness and heterogeneity of generated logs, and in some cases, the need for real-time processing of logs, we anticipate the development of efficient, scalable, and real-time log analysis tools [228].

- We anticipate continued research on current logging practices and log-statement-related issues, as this will enable improvements of future practices and help to create guidelines for developers when making logging decisions. We also predict the evolvement of learning and AI-based log recommender tools and IDE plugins, which utilize the readily available code repositories of open-source projects to provide just-in-time logging practice suggestions to developers [194, 123]. Additionally, we expect further work on logging libraries to collaborate with emerging logging practices and bring in the development of application-specific logging practices [264].

- We foresee automated log file analysis techniques continue to evolve and become more effective and sophisticated (with machine learning and AI-based techniques) in their information extraction from log files and log statements. We also see an emerging trend of new applications that utilize log analysis recently, such as log analysis for code coverage [82]. Moreover, we predict further research will be performed in enabling the analysis of logs for other platforms, such as mobile systems and big-data applications. Log collections will also continue to grow to help with log analysis.

- With regards to log statement prediction, we anticipate that future research on supervised, unsupervised, and deep learning techniques will continue to benefit logs and their analyses.

In this study, we aim to systematically summarize, discuss, and critique the state-of-the-art knowledge in the logging field for experienced researchers, and simultaneously, help new researchers to get a quick and critical grasp of the available research in this area. Additionally, we envision the uncovered research opportunities in this survey serve as a beacon for advancing the logging research.

## 2.8 List of Papers

Table 2.19 provides the list of papers per each category of logging research.

| Topic | Paper title | Year | Venue | Subtopic(s) |
|---|---|---|---|---|
| **(A) Costs and benefits of logging (5)** | Be Conservative: Enhancing Failure Diagnosis with Proactive Logging [324]. | 2012 | OSDI (J) | (E) |
| | Linux Auditing: Overhead and Adaptation [329]. | 2015 | ICC (C) | |
| | Log2: A Cost-Aware Logging Mechanism for Performance Diagnosis [104]. | 2015 | ATC (C) | (E) |
| | A Qualitative Study of the Benefits and Costs of Logging from Developers' Perspectives [192]. | 2020 | TSE (J) | |
| | Log4Perf: suggesting and updating logging locations for web-based systems' performance monitoring [318]. | 2020 | EMSE (J) | (E) |
| **(B) Logging practices (7)** | Characterizing Logging Practices in Open-Source Software [325]. | 2012 | ICSE (C) | (C) |
| | Where do developers log? an empirical study on logging practices in industry [117]. | 2014 | ICSE (C) | (A), (E) |
| | Industry Practices and Event Logging: Assessment of a Critical Software Development Process [250]. | 2015 | ICSE (C) | |
| | Studying the relationship between logging characteristics and the code quality of platform software [279]. | 2015 | EMSE (J) | (D) |
| | Characterizing logging practices in Java-based open source software projects - a replication study in Apache Software Foundation [79]. | 2017 | EMSE (J) | (C) |
| | An Exploratory Study of Logging Configuration Practice in Java [342]. | 2019 | ICSME (C) | (C) |
| | Studying the Use of Java Logging Utilities in the Wild [81]. | 2020 | ICSE (C) | (C), (D) |
| **(C) Logging progression (5)** | An exploratory study of the evolution of communicated information about the execution of large software systems [277]. | 2014 | JSS (J) | (D) |
| | Logging Library Migrations: A Case Study for the Apache Software Foundation Projects [169]. | 2016 | MSR (C) | (B) |
| | Examining the stability of logging statements [170]. | 2018 | EMSE (J) | (B), (D) |
| | Guiding log revisions by learning from software evolution history [196]. | 2019 | EMSE (J) | (D) |
| | Can you capture information as you intend to? A case study on logging practice in industry [265]. | 2020 | ICSME (C) | (B) |
| **(D) Log-related issues (5)** | Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems [322]. | 2014 | OSDI (C) | (B) |
| | Understanding Log Lines Using Development Knowledge [280]. | 2014 | ICSME (C) | (B) |
| | Characterizing and Detecting Anti-Patterns in the Logging Code [78]. | 2017 | ICSE (C) | |
| | Studying and detecting log-related issues [142]. | 2018 | EMSE (J) | (E) |
| | Studying duplicate logging statements and their relationships with code clones [202]. | 2021 | TSE (J) | (C) |
| **(E) Log printing statement automation (15)** | AutoLog: Facing log redundancy and insufficiency [333]. | 2011 | APSys (C) | (A) |
| | Improving software diagnosability via log enhancement [326]. | 2012 | TOCS (J) | (J) |
| | Learning to log: Helping developers make informed logging decisions [346]. | 2015 | ICSE (C) | (B) |
| | LogOptPlus: Learning to optimize logging in catch and if programming constructs [184]. | 2016 | COMPSAC (C) | |
| | Log20: Fully Automated Optimal Placement of Log Printing Statements Under Specified Overhead Threshold [339]. | 2017 | SOSP (C) | (A) |
| | Towards just-in-time suggestions for log changes [194]. | 2017 | EMSE (J) | (C), (D) |
| | Which log level should developers choose for a new logging statement? [193]. | 2017 | EMSE (J) | (B) |
| | SMARTLOG: Place Error Log Statement by Deep Understanding of Log Intention [166]. | 2018 | SANER (C) | (A) |
| | An Approach to Recommendation of Verbosity Log Levels Based on Logging Intention [54]. | 2019 | ICSME (C) | |
| | Which Variables Should I Log? [209]. | 2019 | TSE (J) | |
| | Automatic recommendation to appropriate log levels [175]. | 2020 | SP&E (J) | |
| | Logging Statements' Prediction Based on Source Code Clones [123]. | 2020 | SAC (C) | (B) |
| | Where Shall We Log? Studying and Suggesting Logging Locations in Code Blocks [200]. | 2020 | ASE (C) | (B) |
| | An Exploratory Study of Log Placement Recommendation in an Enterprise System [72]. | 2021 | MSR (C) | (B) |
| | DeepLV: Suggesting Log Levels Using Ordinal Based Neural Networks [203]. | 2021 | ICSE (C) | (B) |

*Snowballing

Table 2.19: A full list of reviewed publications. *'Subtopic'* column shows what other topics are discussed in the research, if applicable.

| Topic | Paper title | Year | Venue | Subtopic(s) |
|---|---|---|---|---|
| **(F) Log maintenance and management (9)** | An integrated data-driven framework for computing system management [199]. | 2010 | TSMCA (J) | (I) |
| | Cloud application logging for forensics [221]. | 2011 | SAC (C) | |
| | FLAP: An end-to-end event log analysis platform for system management [198]. | 2017 | KDD (C) | (I), (J) |
| | Logzip: Extracting hidden structures via iterative clustering for log compression [208]. | 2019 | ASE (C) | (G) |
| | A study of the performance of general compressors on log files [319]. | 2020 | EMSE (J) | |
| | Effective removal of operational log messages: an application to model inference* [285]. | 2020 | arXiv (A) | (I) |
| | Loghub: A large collection of system log datasets towards automated log analytics* [151]. | 2020 | arXiv (A) | (H) |
| | A survey of software log instrumentation [77]. | 2021 | CSUR (J) | (B), (C) |
| | LogAssist: Assisting log analysis through log summarization [210]. | 2021 | TSE (J) | |
| **(G) Log parsing (8)** | What happened in my network: mining network events from router syslogs [258]. | 2010 | IMC (C) | (J) |
| | Baler: deterministic, lossless log message clustering tool [292]. | 2011 | CSRD (J) | |
| | LogSig: Generating System Events from Raw Textual Logs [294]. | 2011 | CIKM (C) | |
| | LogCluster - A Data Clustering and Pattern Mining Algorithm for Event Logs [300]. | 2015 | CNSM (C) | |
| | Spell: Streaming Parsing of System Event Logs [106]. | 2016 | ICDM (C) | |
| | Drain: An Online Log Parsing Approach with Fixed Depth Tree [148]. | 2017 | ICWS (C) | |
| | A Search-based Approach for Accurate Identification of Log Message Formats [225]. | 2018 | ICPC (C) | |
| | Logram: Efficient log parsing using n-gram dictionaries [96]. | 2020 | TSE (J) | (L) |
| **(H) Anomaly detection (15)** | Detecting Large-Scale System Problems by Mining Console Logs* [311]. | 2009 | SOSP (C) | (J) |
| | Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis * [116]. | 2009 | ICDM (C) | (G) |
| | Mining Invariants from Console Logs for System Problem Detection* [212]. | 2010 | ATC (C) | (G) |
| | Linking Resource Usage Anomalies with System Failures from Cluster Log Data [87]. | 2013 | SRDS (C) | (J) |
| | DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning [107]. | 2017 | CCS (C) | (G), (I), (J) |
| | Experience Report: Log Mining using Natural Language Processing and Application to Anomaly Detection [65]. | 2017 | ISSRE (C) | (J) |
| | Execution anomaly detection in large-scale systems through console log analysis* [60]. | 2018 | JSS (J) | (J) |
| | Metric selection and anomaly detection for cloud operations using log and metric correlation analysis* [112]. | 2018 | JSS (J) | (J) |
| | LogAnomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs [224]. | 2019 | IJCAI (C) | (G) |
| | Robust Log-Based Anomaly Detection on Unstable Log Data [337]. | 2019 | ESEC/FSE (C) | (C) |
| | Anomaly Detection via Mining Numerical Workflow Relations from Logs [332]. | 2020 | SRDS (C) | |
| | HitAnomaly: Hierarchical transformers for anomaly detection in system log [160]. | 2020 | TNSM (J) | |
| | LogSayer: Log pattern-driven cloud component anomaly diagnosis with machine learning [343]. | 2020 | IWQoS (C) | (J) |
| | LogTransfer: Cross-system log anomaly detection for software systems with transfer learning [84]. | 2020 | ISSRE (C) | |
| | Semi-supervised Log-based Anomaly Detection via Probabilistic Label Estimation [315]. | 2021 | ICSE (C) | |

*Snowballing

Table 2.19: A full list of reviewed publications (continued). *'Subtopic'* column shows what other topics are discussed in the research, if applicable.

| Topic | Paper title | Year | Venue | Subtopic(s) |
|---|---|---|---|---|
| **(I) Runtime behavior (7)** | An approach for mining web service composition patterns from execution logs [296]. | 2010 | WSE (C) | |
| | Online detection of multi-component interactions in production systems [242]. | 2011 | DSN (C) | (H) |
| | Logmaster: Mining event correlations in logs of large-scale cluster systems [118]. | 2012 | SRDS (C) | (J) |
| | Assisting developers of big data analytics applications when deploying on Hadoop clouds [278]. | 2013 | ICSE (C) | (J) |
| | Performance model derivation of operational systems through log analysis [57]. | 2016 | MASCOTS (C) | (J) |
| | Exploring properties and correlations of fatal events in a large-scale hpc system [102]. | 2018 | TPDS (J) | (J) |
| | Identifying impactful service system problems via log analysis [149]. | 2018 | ESEC/FSE (C) | (J) |
| **(J) Performance, fault, and failure diagnosis (15)** | Assessing and improving the effectiveness of logs for the analysis of software faults [90]. | 2010 | DSN (C) | (I) |
| | Diagnosing the root-causes of failures from cluster log files [88]. | 2010 | HiPC (C) | |
| | SherLog: Error Diagnosis by Connecting Clues from Run-time Logs [323]. | 2010 | ASPLOS (C) | (I) |
| | Detection of software failures through event logs: An experimental study [251]. | 2012 | ISSRE (C) | (B) |
| | Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems* [234]. | 2012 | NSDI (C) | |
| | Failure prediction based on log files using random indexing and support vector machines [115]. | 2013 | JSS (J) | (I) |
| | Leveraging Performance Counters and Execution Logs to Diagnose Memory-Related Performance Issues [291]. | 2013 | ICSE (C) | |
| | lprof: A non-intrusive request flow profiler for distributed dystems* [341]. | 2014 | OSDI (C) | (G), (H) |
| | POD-Diagnosis: Error diagnosis of sporadic operations on cloud applications* [314]. | 2014 | DSN (C) | |
| | Mining system logs to learn error predictors: a case study of a telemetry system [269]. | 2015 | EMSE (J) | |
| | Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs* [321]. | 2016 | ASPLOS (C) | (I) |
| | CRUDE: combining resource usage data and error logs for accurate error detection in large-scale distributed systems [135]. | 2016 | SRDS (C) | (H) |
| | Non-Intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle [340]. | 2016 | OSDI (C) | (I) |
| | Uilog: Improving log-based fault diagnosis by log analysis [349]. | 2016 | JCST (J) | |
| | Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach* [338]. | 2017 | SOSP (C) | |
| **(K) User, business, security, and code coverage (5)** | The unified logging infrastructure for data analytics at Twitter [187]. | 2012 | VLDB (C) | |
| | Identifying recurrent and unknown performance issues [205]. | 2014 | ICDM (C) | (J) |
| | Detection of early-stage enterprise infection by mining large-scale log data [244]. | 2015 | DSN (C) | (H) |
| | The bones of the system: A case study of logging and telemetry at Microsoft [62]. | 2016 | ICSE (C) | (B), (I) |
| | An automated approach to estimating code coverage measures via execution logs [82]. | 2018 | ASE (C) | |
| **(L) Emerging applications (7)** | Operational-log analysis for big data systems: Challenges and solutions [228]. | 2016 | IEEE Softw (J) | (A), (B), (I) |
| | Designing PhelkStat: Big Data Analytics for System Event Logs [274]. | 2017 | HICSS (C) | (H) |
| | Performance evaluation of cloud-based log file analysis with Apache Hadoop and Apache Spark [222]. | 2017 | JSS (J) | (A) |
| | An exploratory study on assessing the energy impact of logging on android applications [86]. | 2018 | EMSE (J) | (A) |
| | Characterizing the natural language descriptions in software logging statements [145]. | 2018 | ASE (C) | (E) |
| | Studying the characteristics of logging practices in mobile apps: a case study on F-Droid [330]. | 2019 | EMSE (J) | (A), (B) |
| | On the naturalness and localness of software logs [126]. | 2021 | MSR (C) | (H) |
| **Total (103)** | | | | |

*Snowballing

Table 2.19: A full list of reviewed publications (continued). *'Subtopic'* column shows what other topics are discussed in the research, if applicable.

# Part III
# Log Prediction

# Chapter 3

# Leveraging Code Clones and Natural Language Processing for Log Statement Prediction

**Abstract-** Software developers embed logging statements inside the source code as an imperative duty in modern software development as log files are necessary for tracking down runtime system issues and troubleshooting system management tasks. Prior research has emphasized the importance of logging statements in the operation and debugging of software systems. However, the current logging process is mostly manual and *ad hoc*, and thus, proper placement and content of logging statements remain as challenges. To overcome these challenges, methods that aim to automate log placement and log content, *i.e.*, *'where, what, and how to log'*, are of high interest. Thus, we propose to accomplish the goal of this research, that is *"to predict the log statements by utilizing source code clones and natural language processing (NLP)"*, as these approaches provide additional context and advantage for log prediction. We pursue the following four research objectives: (RO1) investigate whether source code clones can be leveraged for log statement location prediction, (RO2) propose a clone-based approach for log statement prediction, (RO3) predict log statement's description with code-clone and NLP models, and (RO4) examine approaches to automatically predict additional details of the log statement, such as its verbosity level and variables. For this purpose, we perform an experimental analysis on seven open-source java projects, extract their method-level code clones, investigate their attributes, and utilize them for log location and description prediction. Our work demonstrates the effectiveness of log-aware clone detection for automated log location and description prediction and outperforms the prior work.

**Keywords:**
software systems, software automation, logging statement, logging prediction, source code, natural language processing, NLP, deep learning

**An earlier version of this chapter is accepted for publication in Automated Software Engineering Conference, Doctoral Symposium, 2021 [122].**

## 3.1 Introduction

To gather feedback about computer systems' running state, it is a common practice for developers to insert logging statements inside the source code to have running programs' internal state and variables written to log files. This logging process enables developers and system administrators to analyze log files for a variety of purposes [65], such as anomaly and problem detection [311, 116], log message clustering [217, 300], system profile building [140], code quality assessment [279], and compression of log files [295, 217]. Additionally, the wealth of information in the logs has also generated significant industrial interest and thus has initiated the development of commercialized log processing platforms such as Splunk [43] and Elastic Stack [22].

Due to the free-form text format of log statements and lack of a general guideline, adding proper logging statements to the source code remains a manual, inconsistent, and error-prone task [78]. As such, methods to automate logging ***location*** and predict the ***details***, *i.e.*, the *'static text'* and verbosity level of the logging statement, are well sought after. For example, the log print statement (LPS): log.warn("Cannot find BPService for bpid=" + id), contains a textual part indicating the context of the log, *i.e.*, *description*, *"Cannot find BPService for bpid="*, a *variable* part, *'id'*, and a log *verbosity level*,*'warn'*, indicating the importance of the logging statement and how the level represents the state of the program [15].

For practical concerns such as I/O and development costs, the *quantity*, *location*, and *description* of logging statements should be decided efficiently [166]. Logging too little may result in missing important runtime information that can negatively impact the postmortem dependability analysis [324], and excessive logging can consume extra system resources at runtime and impair the system's performance as logging is an I/O intensive task [339, 104]. In addition, due to the current *ad hoc* logging practices, developers often make mistakes in log statements or even forget to insert a log statement at all [142, 202]. Therefore, prior studies have aimed to automate the logging process and predict whether a code snippet requires a logging statement by utilizing machine learning methods to *train* a

model on a set of logged code snippets, and then *test* it on a new set of unlogged code snippets [117, 346] (supervised learning). A recent work [145] has shown similar code snippets are useful for log statement description (LSD) suggestions by evaluating their *BLEU* [247] and *ROUGE* [206] scores, similar to *Precision* and *Recall*, respectively. Thus, in our research, we specifically seek to utilize source code clones for log statement prediction and suggestion.

Our goal in this research is to utilize code clones as a paradigm to improve the log statement automation task. This will ensure consistency and a higher quality of logging compared to the current developers' *ad hoc* logging efforts. To summarize, the objectives of this research are to first investigate the suitableness of source code clones for log statement prediction, uncover their shortcomings, and then leverage them for automated log location and description prediction based on selecting appropriate source code features [123]. In addition, we utilize deep learning NLP approaches along with code clones to also predict the log statement's description. Through an empirical study of seven open-source software projects, we demonstrate the applicability of similar code snippets for log prediction, and further analysis suggests that log-aware clone detection can achieve high BLEU and ROUGE scores in predicting log statement's description.

## 3.2 Motivating Example

Source code clones are exact or similar snippets of the code that exist among one or multiple source code projects [271]. There are four main classes of code clones [261]: Type-1, which is simply copy-pasting a code snippet, Type-2 and Type-3, which are clones that show syntax differences to some extent, and finally Type 4, which represents two code snippets that are syntactically very different but semantically equal, *e.g.*, iterative versus recursive implementations of *Fibonacci* series in Figure 3.1. In this research, we focus on **method-level code clones** and call the tuple ($MD_i$, $MD_j$) a *'clone pair'*. Figure 3.1 shows that the logging pattern in the original code, $MD_i$ on Line 3 can be learned to suggest logging statements for its clone, $MD_j$, which is missing a logging statement.

**Practical Scenario.** To illustrate how our approach will be useful for developers during the development cycle of the software, we provide the following practical scenario. We consider a possible employment of our research as a recommender tool, which can be integrated as a plugin to code development environments, *i.e.*, IDE software. Alex is a developer working on a large-scale software system and has previously developed method $MD_i$ in the code base. At a later time, Dave, Alex's colleague, is implementing $MD_j$.

```
1 // Original code - MD$_i$
2 int fibonacci(int n){
3  log.info("Calculating Fibo
   sequence for %d.",n)
4  if(n==0||n==1)
5   return n;
6  else
7   return fibonacci(n-1)+
    fibonacci(n-2);
8 }
```

```
1 // Clone Type 4 - MD$_j$
2 int getFibonacci(int n){
3  if(n==0){return 0;}
4  if(n==1){return 1;}
5  int n_2th=0,n_1th=1,nth=1;
6  for(int i=2;i<=n;i++){
7   nth=n_2th+n_1th;
8   n_2th=n_1th;
9   n_1th=nth;}
10 return nth;
```

Figure 3.1: Example for log prediction with code clones.

Our automated log suggestion[1] approach can predict that if this new code snippet, $MD_j$, requires a logging statement by finding its clone, $MD_i$, in the code base. Then, the tool can suggest Dave, just in time, to add a log statement based on the prediction outcome.

## 3.3 Related Work

Prior work has tackled the automation of log statements with various approaches. Yuan *et al.* [324] proposed *ErrLog*, a tool to report error handling code, *i.e.*, *error logging*, such as *catch clauses*, which are not logged and to improve the code quality and help with failure diagnosis by adding a log statement. Zhao *et al.* [339] introduced *Log20*, a performance-aware tool to inject new logging statements to the source code to disambiguate execution paths. *Log20* introduces a logging mechanism that does not consider developers' logging habits or concerns. Moreover, it does not provide suggestions for *logging descriptions*. Zhu *et al.* [346] proposed *LogAdvisor*, a learning-based framework, for automated logging prediction which aims to learn the frequently occurring logging practices automatically. Their method learns logging practices from existing code repositories for *exception* and *function return-value check* blocks by looking for textual and structural features within these code blocks with logging statements. Jia *et al.* [166] proposed an intention-aware log automation tool called *SmartLog*, which uses an *Intention Description Model* to explore the intention of existing logs, and Zhenhao et al. [200] categorized six block-level logging locations.

[1] We use *'suggestion'* and *'prediction'* interchangeably.

Recently, Li *et al.* [202] showed duplicate logging statements that are the outcome of shallow copy-pasting result in log-related anti-patterns (*i.e.*, issues). Although their research has a negative connotation towards copy-pasted logging statements from code clones, it simultaneously shows the potential of code clones as a starting point for automated log suggestion and improvement. In other words, by automating and enhancing the log statements in the clone pairs, we can expedite the development process and avoid shallow copy-pasting that developers tend to do. Additionally, by automation, we reduce the risk of irregular and *ad hoc* developers' logging practices, *e.g.*, forgetting to log in the first place.

Based on the findings of He *et al.*[145] in logging description prediction based on *edit distance* [263], we hypothesize that similar code snippets, *i.e.*, code clones, follow similar logging patterns which can be utilized for log statement location and description prediction. Formally speaking, assuming set $CC_{MD_i}$ is the set of all code clones of Method Definition $MD_i$, if $MD_i$ has a log print statement (LPS), then its clones also have LPSs:

$$\exists LPS_i \in MD_i \implies \forall MD_j \in CC_{MD_i}, \exists LPS_j \in MD_j$$

To evaluate the hypothesis, we guide our research with the following research objectives (**RO**s):

- **RO1:** Demonstrate whether code clones are consistent in their logging statements.
- **RO2:** Propose an approach to utilize code clones for log statement location prediction.
- **RO3:** Provide logging description suggestions based on code clones and deep learning NLP models.
- **RO4:** Utilize clones for predicting other details of log statements such as log verbosity level and variables.

## 3.4 Research Approach

Our research design comprises a preliminary data collection phase, Stage 0, and is followed by four stages, Stages I-IV, to address RO1-RO4, as illustrated in Figure 3.2. In the following, we provide the details of our methodology and current results for each RO.

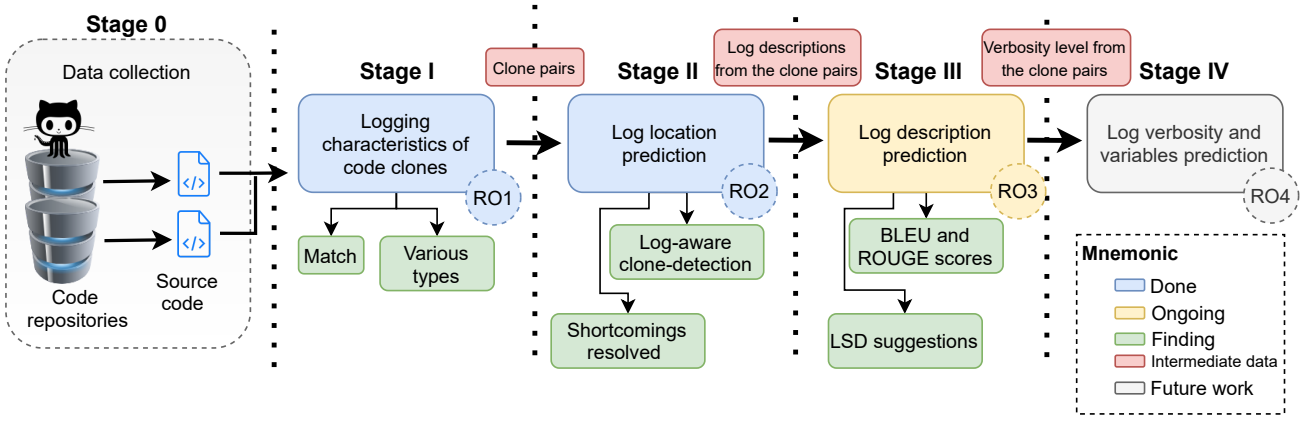



Figure 3.2: Research steps, including objectives, intermediate data, and findings.

### 3.4.1 RO1: Demonstrate whether code clones are consistent in their logging statements and their log verbosity level.

***Motivation.*** To enable code clones for log suggestion, we first require to compare their characteristics and show if clone pairs follow similar logging patterns. ***Approach.*** For this purpose, we select seven large-scale open-source Java projects, *i.e.*, *Apache Hadoop, Zookeeper, CloudStack, HBase, Hive, Camel, and ActiveMQ*, based on the prior logging research [78, 145]. These projects are well-logged, stable, and well-used in the software engineering community, and also they enable us to compare our results with prior work, accordingly. We extract methods with logging statements and then find their clones. ***Evaluation.*** We evaluate the existence of log statements, their verbosity levels, and clone types. ***Results.*** The results show the majority of method clone pairs are consistent in their logging statements and their log verbosity levels also match to a high degree. Additionally, we find that the majority (in the range of 78% to 90%) of code clones are of Types 3 and 4, while the code pairs are matching in the existence of a logging statement. This observation signifies the effectiveness of code clones in suggesting the location of log statements in methods. In other words, although two snippets of clone pairs are syntactically different to a high degree, they still follow similar logging patterns.

### 3.4.2 RO2: Propose an approach to utilize code clones for log statement location prediction.

***Motivation.*** Findings from RO1 show matching logging statements between clone pairs and motivate enablement of logging suggestions with code clones. The automated suggestion approach can help developers in making logging decisions and improve logging practices. ***Approach.*** We initially observe and resolve two shortcomings of general-purpose clone detectors to make them more suitable for log prediction and reduce false positive and false negative cases [124]. We then utilize the clone pairs for suggesting logging statements for the methods which are missing an LPS by finding their clone pairs with a logging statement (Stage II in Figure 3.2). ***Evaluation.*** We evaluate the performance of our approach by measuring *Precision*, *Recall*, *F-Measure*, and *Balanced Accuracy* (BA) on the set of the seven selected projects. ***Results.*** Considering the average of BA values, our log-aware clone detection approach, LACCP, brings 15.60% improvement over Oreo [270] across the experimented projects. With the higher accuracy that LACCP brings, it enables us to provide more accurate clone-based log statement suggestions.

### 3.4.3 RO3: Provide logging description suggestions based on code clones and NLP models.

***Motivation.*** Based on the experiment results for predicting the location of logging statements in RO2 and the additional available context from the clone pairs, *i.e.*, the logging statement description available from the original method, $MD_i$, we notice it is a valuable research effort to explore whether it is also possible to predict the logging statements' *description* automatically. With satisfactory performance, an automated tool that can predict the description of logging statements will be a great aid, as it can expedite the software development process and improve the quality of logging descriptions. ***Approach.*** We base our method on the assumption that clone pairs tend to have similar logging statement descriptions. This assumption comes from the observations in predicting log statements for clone pairs. As logging descriptions explain the source code surrounding them, it is intuitive for similar code snippets to have comparable logging descriptions. Based on this assumption, we propose a deep learning-based method that combines code clones with NLP learning approaches (NLP CC'd). In particular, to generate the LSD for a logging statement in $MD_j$, we extract its corresponding code snippet and leverage LACCP to locate its clone pairs. Laterally, the NLP model provides next word suggestions for the LSDs from the knowledge base available in the training set for each project. ***Evaluation.*** To measure the accuracy of our method in suggesting the log description, we utilize BLEU [247] and

ROUGE [206] scores. These scores are well-established for validating the usefulness of an auto-generated text in prior software engineering and machine learning research, such as comment and code suggestion [50] and description prediction [145]. ***Results.*** We experiment on seven open-source Java systems, and our analysis shows that by utilizing log-aware clone detection and NLP, our hybrid model, (*NLP CC'd*), achieves 40.86% higher performance on BLEU and ROUGE scores for predicting LSDs when compared to the prior research [145], and achieves 6.41% improvement over the No-NLP version [124].

### 3.4.4 RO4: Utilize code clones for predicting other details of log statements such as log verbosity level and variables.

***Motivation.*** Besides log statement location and its LSD, prediction of other details of log statements such as *Log verbosity level (LVL)* and its *variables (VAR)* are useful research efforts and the focus of prior research [193, 54, 203], as they further help the developers in more systematic logging and resolve suboptimal choices of log levels and variables [203]. ***Approach.*** Log-aware clone detection, LACCP, is reasonably extendable to predict LVL and VAR alongside the LSD suggestion. Since we have access to the source code of the method that we are predicting the logging statement for and its clone pair code snippet, a reasonable starting point is to suggest the same LVL as of its clone pair, and then augment it with additional learning approaches such as [193, 54] for more sophisticated LVL prediction. For VAR prediction, our approach can be augmented with deep learning [209] and static analysis of the code snippet under consideration [326] to include log variables suggestions along with the predicted LSD. ***Preliminary evaluation and results.*** Our preliminary analysis for the evaluated projects shows that code clones match in their verbosity levels in the range of (92, 97)%, which confirms that using the verbosity level of the clone pair, $MD_i$, is a good starting point for log verbosity level suggestions for $MD_j$. We are pursuing RO4 as our future work and will provide additional results and findings subsequently.

## 3.5 Discussion

In this section, we compare and discuss the significance of our approach in relation to other existing log prediction and suggestion techniques.

**Method-level log prediction rationale.** Although clone detection (and subsequently, log statement prediction) can be performed in different granularity levels, such as files, classes, methods, and code blocks, however, method-level clones appear to be the

most favorable points of re-factoring for all clone types [178]. We emphasize that our approach also includes all of the logging statements which are nested inside more preliminary code blocks within method definitions, *viz.*, logging statements nested inside code blocks, such as *if-else* and *try-catch*.

**Comparison.** Orthogonal to our research, prior efforts such as [346], [166], and [200] have proposed learning approaches for logging statements' *location* prediction, *i.e.*, *where to log*. The approaches in [346], [166] are focused on error logging statements (ELS), *e.g.*, log statements in *catch clauses*, and are implemented and evaluated on C# projects. Li *et al.* [200] provide log location suggestions by classifying the logged locations into six code-block categories. Different from these works, our approach does not distinguish between error and normal logging statements, is evaluated on open-source Java projects, and leverages logging statement suggestions at method-level by observing logging patterns in similar code snippets, *i.e.*, clone pairs.

**Significance.** Prior approaches [346, 200] rely on extracting features and training a learning model on logged and unlogged code snippets. Thus, they can predict if a new unlogged code snippet needs a logging statement by mapping its features to the learned ones. Although these methods initially appear similar to our approach in extracting log-aware features from code snippets [123], we believe our approach has an edge over the prior work. Because we also have access to the clone pair of the code under development, *i.e.*, $MD_i$ in $(MD_i, MD_j)$, this enables us to obtain and leverage the additional data from $MD_i$ to predict other aspects of log statements, *e.g.*, LSD, which the prior work is unable to do. The significance of our approach becomes apparent in LSD prediction as we utilize the LSD of the clone pair as a starting point for suggesting the LSD of the new code snippet. Thus, our approach not only complements the prior work in providing logging suggestions for developers as they develop new code snippets, but it also has an edge over them by providing additional context for further prediction of LPS details, such as the LSD and the log's verbosity level.

## 3.6 Summary of Contributions

The contributions that become available as the outcomes of our research are as follows: ❶ In RO1, we perform an experimental study on logging characteristics of code clones and show the potential for utilizing clone pairs for logging suggestions. ❷ In RO2, we introduce a log-aware clone detection tool (*LACCP*) [123] for log statements' *'location'* prediction, and resolve two clone detection shortcomings for log prediction and provide experimentation on seven projects and compare it with general-purpose state-of-the-art

clone detector, Oreo [270]. ❸ In RO3, we initially show the natural characteristics of software logs and that enables us to utilize our findings for the application of NLP for LSD prediction [126]. We then propose a deep-learning NLP approach, *NLP CC'd*, to work in collaboration with *LACCP* to automatically suggest log statements' descriptions. We calculate the BLEU and ROUGE scores for our auto-generated log statements' *descriptions* by considering different sequences of LSD tokens, and compare our performance with the prior work [145]. ❹ Finally, as future work in RO4, we investigate the log verbosity level and variables prediction based on the information available through code clone pairs.

Thus far, our research findings have been published for RO1 and RO2 in *ACM Symposium on Applied Computing (ACM SAC)* [123] and *IEEE/ACM Conference on Mining Software Repositories (MSR)* [126], respectively. We have also evaluated the trade-offs associated with the cost of logging statements in our paper accepted in the *International Symposium on Reliable Distributed Systems (SRDS)* [127]. Lastly, the research paper summarizing our contributions for RO3 is currently under review [124].

## 3.7 Conclusions and Future Work

The process of software logging is currently manual and lacks a unified guideline for choosing the location and content of log statements. In this research, with the goal of enhancing log statement automation, we present a study on the location and description of logging statements in open-source Java projects by applying code clones and deep-learning NLP models. We compare the performance of our proposed approaches, LACCP and NLP CC'd, for log location and description prediction, and show their superior performance compared to prior work. As our future work in RO4, we will provide automated suggestions for other details of the LPS, such as its verbosity level and variables.

# Chapter 4

# Code Clones Background

**Abstract-** Source code clone detection is a well-established area of study for software systems, and a significant number of detection techniques and tools have been presented in the literature. Our goal is to illustrate if it is advantageous to utilize code clones for automatic suggestion and prediction of the location and content of logging statements inside the source code. In this chapter, we explain the required definitions and provide the required background to enable source code clones for the purpose of log statement prediction.

**Keywords:**
software engineering; source code; code clones; logging statement; automation

## 4.1 Introduction

Source code clone detection is a well-established area of study for software systems, and a significant number of detection techniques and tools have been presented in the literature [261]. One survey conducted by Rattan et al. [261] listed several clone detection tools existing in the literature. Clone detection is the procedure of locating exact or semantically similar pieces of source code within or between software systems [270] based on metrics of the source code snippets. Clone detection is mostly necessary to detect and prevent copy-paste bugs, maintain software quality, and diminish development costs [271]. Regarding our research, we demonstrate that searching for similar code snippets (*i.e.,* code clones) is beneficial in automated log statement placement, and for this reason, we first present the required background in source code clones.

## 4.2 Source Code Clones

Source code clone detection techniques aim to match exact or similar snippets of code, called clones, among one or more source code projects [271]. The majority of clones are created with code reuse or code copy-paste [267]. Clones may have positive and/or negative impacts. For example, although developers might reuse clones to speed up development time, this can also introduce copy-paste bugs and introduce development uncertainty and cost [267, 266]. Clone management and clone research studies depend heavily on the quality of clone detection tools. In the literature [261], there are four main categories of code clones, ranging from simply copy-pasting a code snippet (i.e., Type 1 in Listing 4.1), all the way up to Type 4 which is two code snippets that are *syntactically* very different but *semantically* equal, e.g., implementing a *sort method* with *bubble sort* versus *merge sort*. The example in Listing 4.1 demonstrates four different types of clones for a function that implements *Fibonacci series*:

1. **Type 1 - exact clones:** two code blocks are exact clones if they are identical with minor syntactic differences such as in layout, comments, or whitespaces and newlines. Lines 1-6 in Listing 4.1 show the original function and Lines 9-14 present a Type 1 clone of the original function.

2. **Type 2 - renamed clones:** two code snippets are renamed clones if they are Type 1 clones with additional modifications such as changes in identifier names, literals values, *etc.* As such, a Type 2 clone is called a renamed clone, i.e., renamed variables (Lines 16-21 in Listing 4.1).

3. **Type 3 - near-miss clones:** these are clones where the copied fragments are similar to the original, however, with modifications such as changes in comments, change of code layout and order of source code elements through blanks and new lines, and changing the identifiers and literals names and types (Lines 23-30 in Listing 4.1). The near-miss name comes from the fact that they are nearly missed by clone detection tools if only looked at syntactical changes of the source code. Very difficult-to-detect clones in this category, meaning clones that although still exhibit few syntactic similarities, and they happen to be extremely hard to detect as clones due to their high degree of syntactic differences, are called *twilight zone clones* [270] or Type 3+.

4. **Type 4 - semantic clones:** two pieces of code are semantic clones such as they implement the same functionally (i.e., similar semantics) but in different syntaxes

(Lines 32-42 in Listing 4.1). Line 37 of Listing 4.1 presents a non-recursive implementation of *Fibonacci* sequence, compared to the original code at Line 1, which presents a recursive version.

## 4.3 Approach

With having the aforementioned types of clones, we hypothesize that if there is a logging statement in the original implementation of a method snippet, $M_O$, then all of the clones of $M_O$ should also contain a logging statement, as they perform similar functionalities. Formally speaking, assuming set $CC_{M_O}$ is the set of all code clones of Method $M_O$, i.e., $CC_{M_O} = \{M_{T1_i}, M_{T2_i}, M_{T3_i}, M_{T4_i}, ...\}$, where $M_{Tx_i}$ is of clone Type x, then:

$$\exists LPS_i \in M_O \implies \forall M_{Tx_i} \in CC_{M_O}, \exists LPS_x \in M_{Tx_i} \tag{4.1}$$

Considering this hypothesis, the idea is if we have an extensive database of logged code, then we should be able to predict if a newly developed code requires a logging statement by looking at its clones in the code database. In the upcoming sections, we validate our hypothesis by an experimental study and predicting if a logging statement is required after removing the logging statement from the test method and then predicting it through its clones. Additionally, code clones have symmetry property; meaning that if $M_{Tx_i}$ is a clone of $M_O$, $M_O$ is also considered a clone of $M_{Tx_i}$. Assuming $\sim_c$ presents a clone relationship, then: $M_O \sim_c M_{Tx_i} \implies M_{Tx_i} \sim_c M_O$; and in a more general term:

$$\forall i, j, \quad M_i \sim_c M_j \implies M_j \sim_c M_i$$

This attribute implies that it does not matter which of the clones is picked as $M_O$. Therefore, we define $(M_i, M_j)$, or equally $(M_j, M_i)$, as a clone pair. We utilize this feature in our experimentation, as for each pair of code clones $(M_i, M_j)$, we initially remove the logging statement for $M_j$, and call it $M'_j$, i.e., $M'_j \sim_c M_j(w/o)_{LPS}$, and aim to predict the existence of a logging statement in $M'_j$, with the condition that if we can still match $M'_j$ with $M_i$ through code clone features. In the next section, we describe our methodology for this study.

## 4.4 Closing Remarks

As we conclude this chapter on the basics of code clones, in the next chapter, we utilize them for log statement prediction.

*Listing 4.1: Four different clone categories.*

```
1  // Original code
2  int fibonacci(int n)  {
3      if(n == 0 || n == 1)
4          return n;
5      else
6         return fibonacci(n - 1) + fibonacci(n - 2);
7  }
8  // Clone Type 1
9  int fibonacci(int n)
10 {
11     if(n == 0 || n == 1)
12         return n; // stop condition
13     else
14         return fibonacci(n - 1) + fibonacci(n - 2);
15 }
16 // Clone Type 2
17 int CalcFibonacci(int i)  {
18     if(i == 0 || i == 1)
19         return i;
20     else
21         return CalcFibonacci(i - 1) + CalcFibonacci(i - 2);
22 }
23 // Clone Type 3
24 int calcFibonacci(int i)  {
25     if(i == 0)
26         return 0;
27     else if (i == 1)
28         return 1;
29     else
30         return fibonacci(i - 1) + fibonacci(i - 2);
31 }
32 // Clone Type 4
33 int getFibonacci(int n){
34     if (n == 0) { return 0; }
35     if (n == 1){ return 1; }
36     int n_2th = 0, n_1th = 1, nth = 1;
37     for (int i = 2; i <= n; i++) {
38         nth = n_2th + n_1th;
39         n_2th = n_1th;
40         n_1th = nth; }
41     return nth;
42 }
```

# Chapter 5

# Logging Statements Prediction Based on Source Code Clones

**Abstract-** Log files are widely used to record runtime information of software systems, such as the time-stamp of an event, the unique ID of the source of the log, and a part of the state of task execution. The rich information of logs enables system operators to monitor the runtime behaviors of their systems and further track down system problems in production settings. Although logs are useful, there exists a trade-off between their benefit and cost, and it is a crucial problem to optimize the location and content of log messages in the source code, i.e., *"where and what to log?"*

Prior research has analyzed logging statements in the source code and proposed ways to predict and suggest the location of log statements in order to partially automate log statement addition to the source code. However, there are gaps and unsolved problems in the literature to fully automate the logging process. Thus, in this research, we perform an experimental study on open-source Java projects and apply code-clone detection methods for log statements' prediction. Our work demonstrates the feasibility of logging automation by predicting the location of a log point in a code snippet based on the existence of a logging statement in its corresponding code clone pair. We propose a Log-Aware Code-Clone Detector (LACC) which achieves a higher accuracy of log prediction when compared to state-of-the-art general-purpose clone detectors. Our analysis shows that 98% of clone snippets match in their logging behavior, and LACC can predict the location of logging statements by the accuracy of 90+% for Apache Java projects.

**An earlier version of this chapter is published in ACM Symposium in Applied Computing (SAC'2020) [123].**

**Keywords:**
software engineering; source code; code clones; logging statement; automation

## 5.1 Introduction

Gathering feedback about computer systems' states is a non-trivial task. For this purpose, it is a common practice to have running programs report on their internal state and variables, through log files that system administrators can analyze [65] for different purposes in order to understand system usage in production and postmortem debugging of system failures. For example, many studies have applied the content of log files to achieve a variety of goals such as anomaly and problem detection [311, 116], pattern detection and log message clustering [217, 300], system profile building, and compression of log files [295, 217]. Although there are methods such as aspect-oriented programming [164] (e.g., AspectJ [164]) to support better modularization of the logging code, many industrial and open-source systems still choose to intertwine the logging code with the feature code [325, 250]. Consequently, the development and maintenance of high-quality logging code as the feature code evolves is critical to the overall quality of software systems [78].

Listing 5.1 shows an example of a *'catch clause'* logged from Apache Hadoop project [27]. In this example, two logging statements exist with different *'severity'* levels on Lines 5 and 8. Log statements can have various *severity* levels based on their importance and how the level represents the state of the program. For example, *fatal* level logging is typically logged prior to a system failure, and *debug* level is logged whenever the system debug mode is active. List of severity levels for Log4j [15], an Apache logging library, includes: *trace, debug, info, warn, error,* and *fatal*. Additionally, each logging statement contains a *static text* (e.g., *"Authentication exception:..."* on Line 5) and possible variables (e.g., *ex.getMessage()*). This logged information in the *'catch clause'* will be useful at the time of debugging in case an exception happens and for failure detection.

Although logs have proven to be useful, adding proper logging statements to the source code remains a manual, arbitrary, and in some cases an error-prone task [191] due to the free-form text format of logging statements and lack of a general guideline. Besides, for practical concerns, the number of logging statements cannot be too few nor too many [166]. On the one hand, logging too little results to miss important runtime information that can negatively impact the postmortem analysis [324]. On the other hand, too many logging points can consume extra system resources at runtime, and impair the system performance, as logging is an I/O intensive task. Therefore, thoughtful log placement methods aim to

*Listing 5.1: Catch clause logging example from Apache Hadoop.*

```
1  catch (AuthenticationException ex) {
2    errCode=HttpServletResponse.SC_FORBIDDEN;
3    authenticationEx=ex;
4    if (LOG.isDebugEnabled()) {
5      LOG.debug("Authentication exception: " + ex.getMessage(),ex)
       ;
6    }
7   else {
8      LOG.warn("Authentication exception: " + ex.getMessage());
9    }
10 }
```

record valuable information at *log points of interest* systematically, and, simultaneously, not to introduce unnecessary system overhead.

Considering the trade-offs as mentioned above for logging, our goal is to perform an experimental study to enable predicting the location of log statements based on source code features in order to automate log statement placement. Prior recent studies have introduced methods and tools for partially automating the logging process and/or optimizing the number of log statements. For example, proposed machine learning methods [117, 346] predict whether or not a code snippet needs a logging statement by training a model on a set of logged code snippets, and testing it on a new unlogged code set, i.e., *supervised learning*. However, prior researches failed to propose a general solution based on an arbitrary code snippet, as they mostly concentrate on error log statements such as *exception handling logging* [346]. Thus, for introducing a general approach for log prediction and automation, we consider the application of code clone detection methods. Code clone detection is the task of locating syntactically exact or similar snippets of source code (with equal semantics) within or between software systems based on contextual metrics of the source code snippets [270]. Our goal is to predict the location of logging statements within the source code. As we will show in the following, searching for similar code snippets (i.e., code clones) can be beneficial in automating the logging statement generation, and for this reason, we conduct an experimental study on code clones. We then utilize code clones for logging statements prediction.

To summarize, the key contributions of this research are:

- We perform an experimental study of logging practices in open-source software systems, and investigate the possibility of using source code clones for predicting the location of log statements.

- We introduce a set of Research Objectives (ROs) to help us formulate log location prediction, which we believe is the first work to consider code clones for log location prediction. Based on the ROs, we then propose our findings in order to utilize code clones for log statements' prediction and suggestion.
- We show the feasibility of log statement prediction with a set of experiments by removing the log statements and then predicting them by utilizing their code clones.
- We propose our log-aware code clone detection tool (LACC) which significantly increases log prediction accuracy when compared against another state-of-the-art clone detection tool [270].

The rest of this chapter is organized as follows. Section 5.2 describes the related work for logging prediction and code clone detection. Section 5.3 provides the necessary definitions and our approach for logging statement prediction. In Section 5.4, we explain our study methodology and discuss the pertinent research objectives. Then, Section 5.5 explains our findings from the experimental study, and we introduce LACC in Section 5.6. In Section 5.7, we mention valid threats to the generalization of our research. Finally, we present our conclusions and future directions in Section 5.8.

## 5.2 Related Work

In this section, we discuss the prior work related to our research. In our work, we perform an experimental study to predict the location of logging statements by utilizing code clones. Therefore, we discuss related work along the lines of the empirical analysis of log statements, logging prediction research, and code clone detection approaches.

### 5.2.1 Empirical Analysis of Log Statements

Several prior research efforts have explored different logging practices in open-source as well as proprietary software projects. Yuan *et al.* [325] studied logging practices in open-source software systems. It appeared to the authors that software developers spend a significant amount of time modifying log messages whenever logging issues arise. To facilitate this concern, they proposed a simple checker to detect logging issues automatically. In a follow-up study, Yuan *et al.* proposed *LogEnhancer* [326], a tool to enhance logging statements by adding additional information to them (e.g., live variables) in order to improve postmortem

analysis. Fu *et al.* [117] analyzed 100 randomly chosen log statements from two proprietary software systems from Microsoft written in C#. They classified the log statements into five usual categories of logging among them *exception handling*, *assertion-check values*, and *function return-value* checks. They further performed a survey among Microsoft employees on *where* do developers put logging statements. In contrast to these studies, our research focuses on finding code clone features that can help an automatic logger to predict the location of source code logging statements.

### 5.2.2 Logging Statement Prediction

Yuan *et al.* proposed a tool, *ErrorLog* [324], to report error handling code (e.g., *'catch clause'*) that is not logged. Then, they would add logging statements to the reported unlogged catch clauses to improve them and facilitate failure diagnosis. ErrorLog logs exception code blocks in C# projects, as a partial step towards automating logging statements. Later on, Zhu *et al.* [346] proposed *LogAdvisor*, a learning-based framework, for logging prediction which aims to automatically learn the developers' common logging practices. Their method learns logging practices from existing code repositories for *'focused code snippets'*, i.e., *'exception'* and *'function return-value check'* code blocks. LogAdvisor looks for and extracts textual and structural features within these types of code blocks with logging statements. As such, LogAdvisor only focuses on exception handling and function return-value logging which is rather used for error handling than being a general approach. In a recent work, Lal *et al.* [185] proposed a feature based method for logging prediction within *'catch'* and *'if-else'* blocks. Our research is different from these works as we consider logged methods, which is a more generic approach, instead of a particular code block like catch clause which focuses on *error logging*. Moreover, we apply clone detection for identifying similar logging patterns, which we believe is the first work to do so.

### 5.2.3 Code Clone Detection

Source code clone detection is a well-established area of study for software systems, and a significant number of detection techniques and tools have been presented in the literature [261]. Clone detection is the procedure of locating exact or semantically similar pieces of source code within or between software systems [270] based on metrics of the source code snippets. Clone detection is mostly necessary to detect and prevent copy-paste bugs, maintain software quality, and diminish development costs [271]. Regarding our research, we demonstrated that searching for similar code snippets (i.e., code clones) is beneficial in

automated log statement placement, and for this reason, we conducted an experimental study on code clones for log statement prediction purposes. Additionally, we improved the accuracy of logging statements prediction with log-aware clone detection (LACC).

## 5.3 Definitions, Background, and Approach

In this section we first review the required definitions for this work followed by our approach for applying code clones.

### 5.3.1 Definitions

**Code Blocks**. A program's source code consists of a set of code blocks, $CB = \{cb_1, cb_2, ..., cb_n\}$, such as *method definition, if-else, switch-case, while-loop, etc.*

**Log Print Statement (LPS)**. A log print statement records an event that occurs in the running software with a *severity level*, a *static text* (i.e., *description*), and optional *variables* (Listing 5.1).

**Log Point of Interest (LPI)**. A point of interest for logging is defined as a place in the source code that has been commonly logged by the developers, and that point follows a particular pattern. For example, many developers tend to put an error log message inside the *"try-catch"* block. In this work, we consider the developers chosen points of logging as *ground truth* and evaluate the accuracy of our prediction by comparing with these points. In this research, we consider LPIs at the method level, i.e., logging statements that exist inside method definitions code blocks ($CB_m$). This approach also includes all of logging statements which are nested inside more preliminary code blocks within method definitions, such as logging statements nested inside code blocks such as *if-else*, *try-catch*, *etc.*

**Method Log Placement (MLP)**. A MLP placement, $CB_m$, is a subset of $CB$ where at least one LPS exists in each method code block. In our research, considering log prediction at method-level LPI, we consider $CB_m \subset CB$ as the set of all method definitions with at least one logging statement in their body. With the definitions mentioned above, we continue with the problem definition and our solution:

**Problem**. To automate the log statements placement and predict the proper set, $CB_m \subset CB$, i.e., the set of method definitions with a logging statement, in the software systems source code.

**Solution**. For this purpose, we first perform an experimental study on the attributes of method-level code blocks containing a logging statement by use of code clone detection tools. Then, we define pertinent research objectives (ROs) regarding the logging behaviour, in the sense of consistency of logging statements among code clones, and whether or not we can utilize this information for logging automation and prediction. Later on, we predict the existence of logging statements in methods with having their logging statements removed.

### 5.3.2 Source Code Feature Formulation

In order to be able to predict log statements, we first need to define pertinent source code features, which we utilize to predict whether a code block requires a log statement. We use these features to feed a machine learning tool (Figure 5.1) in order to identify similar code snippets (i.e., clone pairs), which we argue should follow similar logging patterns.

*Functional* and *structural* features are the two main categories of source code features. For example, many developers tend to put an error-log message inside *"try-catch"* blocks. The *try-catch* block is an example of a *structural* feature. On the other hand, *functional* features are concerned with modules of the system and how they interact with each other. For example, a code block, $cb_i$, might be logged as other modules have frequently referenced to it. *Functional* and *structural* features can be quantitatively measured by classifying them based on their types: *boolean*, *numerical*, and *string* features.

Let $F$ be the set of all features for the source code under consideration, i,e., F = $\{f_1, f_2, ..., f_n\}$. For example, Table 5.2 lists several log-related features for our analysis. Now, let us assume $LPS(cb_i)$ to present the existence of a log statement (LPS) in a given block, $cb_i$. As we mentioned earlier, we focus on *method definition* code blocks, and we extract the methods that contain logging statements, i.e., $LPS(cb_M) == 1$. With this introduction, we define *boolean*, *numerical*, and *string* features as follows:

#### Boolean features

A boolean feature has values of 0 or 1. If Feature $f_n$ exists in Block $cb_i$ then:

$$Fr_B(cb_i, f_n) = \begin{cases} 1, & \text{if } f_n \text{ exists in block } cb_i \\ 0, & \text{otherwise} \end{cases}$$

For example, considering the existence of a logging statement in a method definition, if there is a logging statement in Block $cb_i$ then: $Fr_B(cb_i, f_{LPS(cb_i)}) == 1$.

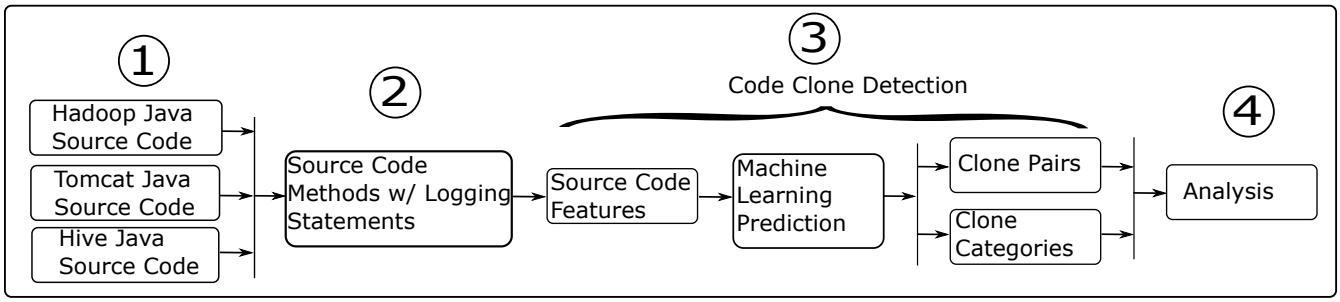



Figure 5.1: Methodology for the experimental study.

**Numeric features**

Let $n_{f_n}(cb_i)$ be the count of occurrence of Feature $f_n$ that exists in the code Block $cb_i$, therefore:

$$Fr_N(cb_i, f_n) = \begin{cases} n_{f_n}(cb_i), & n_{f_n}(cb_i)>0, \text{ the count of occurrence of } f_n \text{ in block } cb_i \\ 0, & \text{otherwise} \end{cases}$$

Note that $Fr_N(cb_i, f_n)$ can be larger than 1, as Block $cb_i$ can have multiple occurrence of this feature. For example, considering the number of statements (NOS) in the method body ($M_i$) in Listing 5.2 as the feature, it has eight statements, i.e., $Fr_N(M_i, NOS) == 8$.

**String features**

These features include log related keywords, such as log function wrappers for Log4j [15], e.g., *log.info(*)* and *logger.info(*)*, and log severity levels, e.g., *error, debug, info, fatal, etc.* These features help with our log-aware clone detection, as we can distinguish the existence of log statements and calculate specific metrics, such as the number of tokens, NTOK in Table 5.2, based on the number of logging statement that exists in the method body. In the next section, we discuss code clones and how we utilize them for logging prediction.

## 5.4 Study methodology

This section consists of the toolchain steps and our algorithm for log prediction. The details of each step are as follows.

### 5.4.1 Toolchain

Figure 5.1 illustrates the four steps in our toolchain for log statement prediction:

1. In Step 1, we select and obtain open-source Java projects from their Git repositories. We picked the projects based on factors of interest such as the age and size of the project (in the number of source code lines), popularity (being well-established) and stability of the project, and logging index of the project [78]. After inquiring the long list of available projects, we selected three Apache Java projects: Hadoop, Tomcat, and Hive.

2. In Step 2 of Figure 5.1, we extract method definitions containing logging statements by applying JavaParser [297]. Initially, we parse the source code to its abstract syntax tree (AST) which is the hierarchical representation of the code. We use the AST of the source code to access Java method definitions. We look for methods with a logging statement in the tree, and then pass them to Step 3 for clone detection. Referring to Formula 4.1, in Step 2, we extract all of the $M_O$ methods which have at least one LPS in their method body.

3. In Step 3, we perform code clone detection [270] on the extracted code snippets ($M_O$ methods) from Step 2. We briefly review clone detection steps in the following. Initially, we apply SourceCC [271] and Oreo [270], which are the state-of-the-art general purpose code clone detection tools for code clone analysis on the extracted $M_o$s in Step 3 of Figure 5.1. Oreo is capable of detecting clones up to Type 3+ which enables us to compare code clones that contain a logging statement more accurately. Clone detection tool extracts source code features [270] from the method definition such as McCabe's cyclomatic complexity [223], number of statements, number of expressions, *etc.* These features are used for detecting Type 1 and 2 clones. In order to detect Type 3 and higher-level clones, [270] uses a deep learning (DL) model trained on a massive set of code snippets (50k Java projects from Github). This extensive training set minimizes the risk of overfitting [75] for the testing set. The DL model predicts if two code snippets are clones or not, which is then used for our log statement analysis. We review more details of the DL model training in Section 5.5.6. The output of Step 3 is the set of all clones for methods with logging statements. Later on, in Section 5.6, we update source code features to be log-statement-aware and achieve a higher prediction accuracy.

4. Finally, in Step 4, we perform our analysis on the results of clone detection and collect statistics on the clone types and log severity levels of the code clones. We

also investigate if code clones have similar logging behaviour (i.e., the presence of a logging statement in both clone pairs), as well as if their logging severity levels matches or not (e.g., whether or not both clone snippets have info level or not). We also perform experimentation on the prediction accuracy of logging statements.

### 5.4.2 Algorithm

Algorithm 1 summarizes our procedure for log prediction. We initially parse the source code files from Java Git repositories to their AST representations (Line 1). We define $CC_{M_o}(pairs)$ as the set of all clone pairs of Method $M_o$, i.e.,: $CC_{M_o}(pairs) = \{(M_o, M_{Tx_i})|\forall M_{Tx_i} \in CC_{M_o}\}$. In Line 2, we extract all method definitions with at least one LPS (log print statement) from the AST. Then, in the *for-loop* on Lines 4-9, we initially find all clone pairs for each method definition $M_o$. Then, we create clone pairs of $(M_o, M_{Tx_i})$, and add them to the set of all clone pairs, $Clone_{pairs}$. In the *for-loop* on Lines 7-9, for each clone pair $(M_i, M_j)$, we initially remove the logging statement from $M_j$ and name it $M'_j$. Then, we try to find clones of $M'_j$ on Line 11. If we are successful to detect $M_i$ as a clone pair for $M'_j$, i.e., $M'_j \sim_c M_i$, then we increment *logPredicted* on line 12. Otherwise, *misPredicted* is incremented on line 14. In the set of $M_{logPred}$, we keep track of all $M'_j$s which have been matched, along with the corresponding logging print statement from their corresponding clone pairs, $M_i$s. This information can be used at a later time for suggesting a logging statement for $M'_j$ based on the logging statement retrieved from its clone, $M_i$, i.e., $LPS(M_i)$, which we keep track in $M_{logPred}$ set. Finally, on Line 16, the algorithm returns the accuracy of the prediction and $M_{logPred}$, the set of $M'_j$s with the proposed logging statements, $LPS(M_i)$.

An example of how this approach is useful for logging automation is that: "assume $M_i$ has been previously developed in the code base. Now, another developer is implementing $M'_j$ at a later time. An automated log suggestion tool can predict that if this new snippet of code, $M'_j$, needs a logging statement by finding its clone $M_i$ in the code base. Then, the tool can suggest to the developer to add a log statement based on the prediction outcome. It can even suggest the static text for the logging by suggesting the text existing in its clone log statement". In the next section, we discuss our research objectives as a roadmap for logging statement prediction.

**Algo. 1:** Log Statement Predictor

**Input:** Java source code repositories

**Output:** Log statement prediction, prediction accuracy

$sourceCode_{AST} \leftarrow Parse(sourceCode)$;

$Methods_{LPS} \leftarrow extract(sourceCode_{AST}, exist(LPS))$;

$clone_{pairs} \leftarrow \{\}$;

**for** $(\forall M_o \in Methods_{LPS})$ **do**

  $CC_{M_o} \leftarrow$ `findClones(`$M_{Ti_x}$`)` $| M_{Ti_x} \sim_c M_o$;

  $CC_{M_o}(pairs) \leftarrow createPairs(CC_{M_o})$;

  $clone_{pairs} \leftarrow CC_{M_o}(pairs) \bigcup clone_{pairs}$;

$logPredicted \leftarrow 0$;

$misPredicted \leftarrow 0$; $M_{logPred} \leftarrow \{\}$;

**for** $(\forall (M_i, M_j) \in clone_{pairs})$ **do**

  $M_{j'} \leftarrow removeLPS(M_j)$; **if** $M_i \in$ *findClones*$(M_{j'})$ **then**

    $logPredicted$**++**;

    $M_{logPred} \leftarrow M_{logPred} \bigcup (M'_j, LPS(M_i))$; **else**

      $misPredicted$**++**;

$$Accuracy = \frac{logPredicted}{logPredicted + misPredicted};$$

**return** $M_{logPredicted}, Accuracy$;

### 5.4.3 Research Objectives on Clone Detection for Logging Statement Prediction

In order to utilize code clones for logging prediction, we perform an experimental study on logging statements in open-source software systems. We examine large and well-established open-source projects and pursue three research objectives (ROs). As explained, our focus is on the methods of the source code that contain at least one logging statement. Then, we can identify method code clones and review their logging characteristics. We plan out the following research objectives:

1. **RO1:** demonstrate that code clones are consistent in their logging statements and their log severity level (e.g., does a clone pair apply a matching *info* or *warn* log severity level?).

2. **RO2:** extract the categories of code clones with logging statements (i.e., Type 1, 2,

or 3+).

3. **RO3:** apply method level code clone detection for logging statement placement prediction.

In the following section, we pursue answers for the ROs mentioned above and explain how these help in logging statement prediction and automation.

## 5.5 Experimental Study

In this section, we conduct experiments on the proposed ROs as well as the accuracy of logging statements prediction. First, we explain the rationale behind selecting method-level logging prediction and review the characteristics of the systems under study.

### 5.5.1 Method-level clone detection and logging prediction

Clone detection can be done in different levels of granularity, such as *files, classes, methods, if-else blocks, statements,* or even *sequences of source lines*. However, *method-level* clones appear to be the most favourable points of refactoring for all clone types. They tend to have a considerable amount of code in common, and are the meaningful clones which are also useful for software maintenance and evolution phases [178]. Therefore, in this research, we consider LPIs at the method level, i.e., logging statements that exist inside method definitions code blocks ($CB_m$). We emphasize that our approach also includes all of the logging statements which are nested inside more preliminary code blocks within method definitions, such as logging statements nested inside code blocks such as *if-else*, *try-catch*, *etc.*

### 5.5.2 Systems under study

Based on the prior research on open-source projects in [78], we selected three open-source Java projects (i.e., Apache Hadoop, Tomcat, and Hive) that are considered logged intensive and are stable and well-used in the software engineering community. Table 5.1 summarizes the line number of source code and the number of logging statements in each project. The last row in Table 5.1 presents the number of logs per thousand lines of code (KLOC). All of these projects use Apache Log4j library [15] for logging statements, which includes 6

log severity levels: *fatal, error, warn, info, debug, and trace*. We observed that although all these projects are from the Apache umbrella, they are from different domains and are developed by different teams, and cover different logging practices. In the following, we review and explain our findings concerning each RO.

| Statistics \ Projects | Hadoop | Tomcat | Hive |
|---|---|---|---|
| total lines of code | 2.10M | 0.96M | 1.60M |
| # of log statements | 16,202 | 5,215 | 8,312 |
| # of log statements per KLOC | 7.72 | 5.44 | 5.20 |

Table 5.1: Selected projects statistics for the experimental study.

### 5.5.3 RO1: demonstrate that code clones are consistent in their logging statements and their severity level.

#### Approach

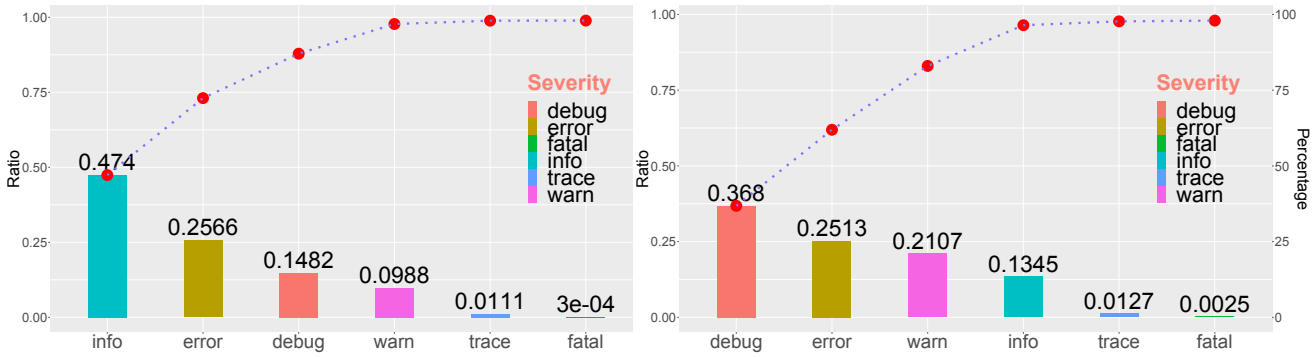
for this RO, we initially analyze the source code of the selected projects and extract the clone pairs, as plotted in Figure 5.1 and explained in Algorithm 1. Then, for each project, we calculate the distribution of logging statements severity levels for the clone pairs.

#### Outcome

in Figures 5.2a and 5.2b, the left vertical axes show the ratio of the number of each log severity level (i.e., info, debug, *etc.*) to the total number of logs and the secondary vertical axis (on the right) presents the accumulative percentage of logs in each category. For example, in Figure 5.2a, 47.4% of the logging statements in the Hadoop project are of *info* severity level and clone pairs are matching in their severity level, i.e., for $clone_{pair}$ $(M_i, M_j)$: `if` $M_i(LPS_{info}) \implies M_j(LPS_{info})$, or generally speaking, for severity level $sl$:

$$\texttt{if}\ M_i(LPS_{sl}) \implies M_j(LPS_{sl})$$

Figures 5.2a and 5.2b illustrate, for Hadoop and Tomcat respectively, that accumulatively over 98% (by adding all the values on the figure and then ratio * 100) of the clones




(a) Log severity level breakdown for Hadoop. (b) Log severity level breakdown for Tomcat.

Figure 5.2: Log severity level breakdown for Hadoop and Tomcat projects.

match in their log severity. We calculate the accumulative value by adding up the percentage value of all the severity levels. Therefore, only in ∼2% of the log statements the severity level differs in clone pairs, plotted in Figure 5.3. We omitted Hive log severity level breakdown due to limited space, and as we likewise observed similar trends for Hive clone pairs with *error* severity level being the highest contributor. This observation signifies that code clones can be used to predict the severity level of logging statements from one code clone to its clone pair. One application of this observation is to automatically suggest the logging severity of a logging statement in a newly developed snippet of code, by searching for its clone pairs in the repository.

**Log severity level mismatch:** here mismatch means that both clone pairs have a logging statement but the severity level is inconsistent, e.g., $M_i$ uses *error* severity level and $M_j$ uses *warn* level instead. Figure 5.3 illustrates, for Hadoop, Tomcat, and Hive that, respectively, only 1.11%, 2.03%, and 0.06% of the clone pairs mismatch in their log severity level. Therefore, 98% or higher percentage of the clone pairs follow the same log severity levels in their logging statements.

**Finding 1: clone pairs are consistent in the existence of logging statements in them, and we can use the clones to predict the severity level of logging statements with a high accuracy.**

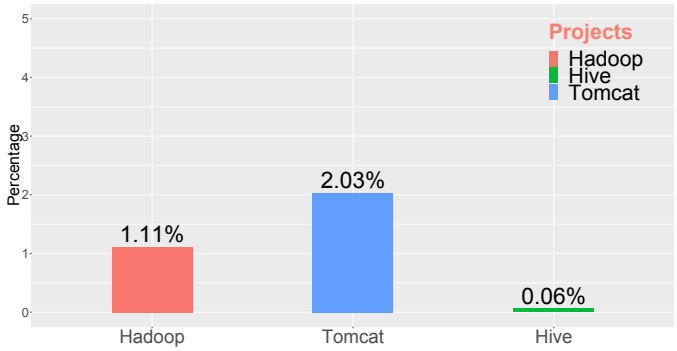



Figure 5.3: Log severity level mismatch for three Java projects.

### 5.5.4 RO2: extract the categories of code clones with logging statements.

#### Approach

After the clone pairs are extracted, we look into each clone pair to determine their clone types. We apply Levenshtein distance [190] in conjunction to the output of Step 3 in Figure 5.1 to categorize clones into different types. In this approach, the entire method code snippet (i.e., method's name and method's body) is treated as a string, and we calculate the difference using character-based edit distance between $M_i$ and $M_j$. For example, the Levenshtein distance between *"int fibonacci(int n)"* and *"int calcFibonacci(int i)"* is 6, as it requires 6 substitutes to make them identical.

#### Outcome

In Figure 5.4 the vertical axis shows the ratio of the number of each clone type (i.e., Type 1, 2, and 3+) to the total number of code clones for all three projects, i.e., Hadoop, Tomcat, and Hive. From Figure 5.4 we can observe that the majority (in the range of 78% to 90%) of code clones are of Type 3 or beyond (3+), and the code clones are matching in the existence of a logging statement. Only a small portion of clones are Type 1 and 2. This observation signifies the usefulness of clone detection in distinguishing the location of log statements. In other words, although two snippets of clone pairs are syntactically different to a high extent in clone Type 3 and beyond (3+), but we can still match their logging behavior, which is beneficial in predicting the location of a logging statement in an unlogged code snippet from its clone pairs in the code base.

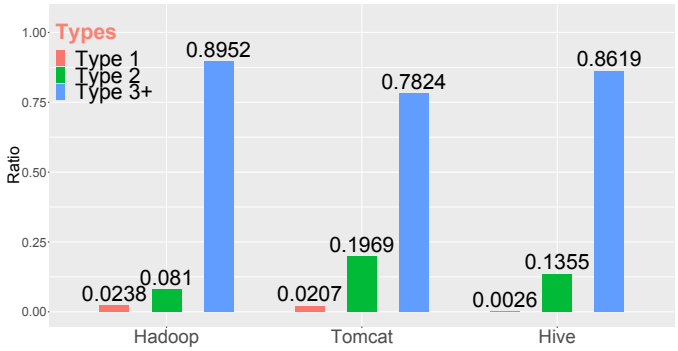



Figure 5.4: Clone types breakdown for Hadoop, Tomcat, and Hive.

**Finding 2: although code clones syntaxes can become significantly different (*i.e.*, Type 3+), they still follow similar patterns of logging statements.**

### 5.5.5 RO3: apply method level code clone detection for logging prediction.

From previous ROs, we can infer that code clones are helpful in the prediction of logging statements location, as well as their severity level. These findings can enable us to predict and suggest logging statements automatically. As we explained briefly before, a practical scenario can be such as assuming a developer is developing a new snippet of source code. Being able to identify clone pairs of this new code snippet as the developer is entering it by looking into the code base, can be beneficial to suggest to the developer if a logging statement is recommended. This scenario can be even extended to multiple source code projects, and bring intra- and/or inter-project logging suggestions. The latter is useful for small projects in which there is not a sufficient prior code base, which is commonly referred to as the project's *"cold start"* phase.

### 5.5.6 Log Prediction

In this experiment, we initially run the non-modified clone detection tool (i.e., Oreo [270]) on the source code base of three projects. In order to ensure variability in the dataset, the machine learning engine of the clone detector is trained on 50K randomly selected Java projects from Github [270]. Randomly, 80% of each project is selected for training and

the remaining 20% for testing. Additionally, one million pairs from the training set of 50K projects are kept separately for validation to avoid bias and overfitting [75]. With this setup, the authors of [270] showed that Oreo outperforms every other compared tool in the accuracy of clone detection. We use this model on our testing set (Hadoop, Tomcat, and Hive projects) such that for each detected code clone pair of $(M_i, M_j)$, we initially remove the logging statement for $M_j$ and call it $M'_j$. Then, we aim to predict the existence of a logging statement in $M'_j$ if we are still able to match $M'_j$ with $M_i$ through code clone features. The result of this experiment is shown in Figure 5.5. Gray bars represent the accuracy of log statement prediction for three different projects, which is in the range of $(43\%, 97\%)$ by applying the state-of-the-art general-purpose clone detection tool, Oreo [270].

### 5.5.7 A Clone Detection Shortfall

Note that, in the experiment in Section 5.5.6, we used the same source code features as the general-purpose clone detection tool uses [270]. By manually investigating some of the clone mismatches, we noticed the main cause for low prediction accuracy in some cases (e.g., in the Tomcat project) is that the eliminated logging statement(s) contributed to a significant portion of the method body, and therefore, after their removal, a method's source code features significantly changed which resulted in a mismatch from the clone detector. For example, in Listing 5.2, Methods $M_i$ and $M_j$ are initially considered clone pairs. However, after removing four logging statements from $M_j$, on Lines 4, 6, 7, and 9, and resulting in $M'_j$, $M'_j$ becomes significantly different when compared to $M_i$, and therefore, it becomes difficult for general-purpose clone detection techniques [270] to capture the similarity of the semantics excluding the logging behavior. Therefore, we propose a log-aware clone detection technique by updating source code features.

## 5.6 Log-Aware Code Clone Detector (LACC)

As we noticed, the main cause for average accuracy for some projects in Figure 5.5 is that the eliminated logging statement(s) contributed to a significant portion of the method's body, and therefore, after their removal, a method's source code features significantly changed which resulted in a mismatch (Listing 5.2). To be more accurate, we also discard the surrounding code if their sole purpose is for adding a condition for logging, such as *if blocks* on Lines 18 and 20 in Listing 5.2. In order to achieve a log-aware clone detection with higher prediction accuracy, we have remodelled the features in Table 5.2 in the clone

*Listing 5.2: Clone mismatch example after removing log statements.*

```
1  //M_i
2  public Exception doAbortedPOSTTest(AbortedPOSTClient client, int
    status, boolean swallow) {
3      Exception ex = client.doRequest(status, swallow);
4      if (log.isDebugEnabled()) {
5       log.debug("Response line: " + client.getResponseLine());
6       log.debug("Response headers: " + client.getResponseHeaders
        ());
7       log.debug("Response body: " + client.getResponseBody());
8       if (ex != null) {
9        log.info("Exception in client: ", ex);
10      }
11     }
12     return ex;
13 }
14
15 //M'_j; i.e., M_j after removing logging statements
16 public Exception doAbortedUploadTest(AbortedUploadClient client,
    boolean limited, boolean swallow) {
17  Exception ex = client.doRequest(limited, swallow);
18     //if (log.isDebugEnabled()) { //logging-dependent code is
       also disregarded.
19          //logging statements removed
20          //if (ex != null) {
21           //logging statement removed
22          //}
23     //}
24     return ex;
25 }
```

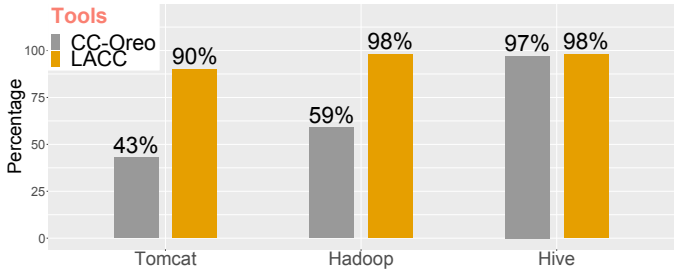



Figure 5.5: Percentage of log prediction accuracy.

detection algorithm to understand and capture log statements and consider them appropriately for the case of clone detection. As such, we introduce LACC for log-aware clone detection. In Table 5.2, *SLOC* includes all lines of the source code (such as comments, brackets({}) for *if-else* blocks, *etc.*; however, *NOS* includes only executable expressions in the method definition.

By introducing the features in Table 5.2 to be log-aware, we modify the clone detection phase to understand log statements, and incorporate accordingly in calculating these features such that methods are detected as clone pairs regardless of the presence of logging statements. Concisely, we perform the following steps for LACC:

- Scan the source code AST and locate methods with logging statements, $M_i$s.
- Recalculate the features based on the existence of the logging statements and logging-related code (Listing 5.2). Features are updated in a way that methods are detected as clone pairs regardless of the presence of logging statements, i.e.,
  $Fr(M_i) \sim Fr(M_j) \implies Fr_{updated}(M_i) \sim Fr(M'_j)$.
- As LACC performs log-aware feature calculation, it understands the log changes while processing the source code, and it achieves a higher log prediction accuracy.

Figure 5.6 partially depicts LACC, which is similar to our toolchain presented in Figure 5.1. The green box has been updated to incorporate log-aware feature calculation. For example, in Listing 5.2, removing {*log.debug("Response line: " + client.getResponseLine())*}, causes the number of tokens for $M'_j$ (i.e., $NTOK_{M'_j}$) to decrease by 5 when compared to $NTOK_{M_i}$. Similarly, $SLOC_{M'_j}$ and $NOS_{M'_j}$ values decline as an executable line of $M'_j$ has been removed. $LMET_{M'_j}$ and $XMET_{M'_j}$ values also potentially decrease as the omitted

| Feature | Description |
|---|---|
| NOS | Number of method statements |
| NEXP | Number of expressions |
| NTOK | Number of tokens |
| LMET | Number of local method calls |
| XMET | Number of external method calls |
| SLOC | Source lines of code |

Table 5.2: Method-level log related features.

line makes references to other methods. The more logging statements exist in Method $M_j$, the more source code features will diverge for $M_i$ and $M'_j$ after log statements are removed. With the LACC design, we accommodate for these feature changes accordingly.

Yellow bars in Figure 5.5 show the accuracy of our tool, LACC, for log statement prediction for three different projects, which is in the range of $(90\%, 98\%)$ and significantly outperforms CC-Oreo [270], the gray bars in Figure 5.5.

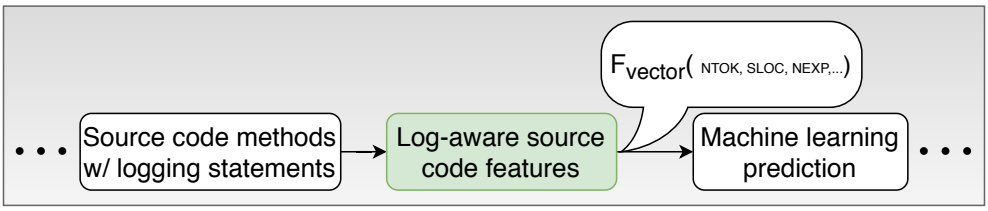



Figure 5.6: LACC updated section.

**Finding 3: we can predict and suggest the location and severity level of logging statements in the source code with high accuracy in the range of** $(90\%, 98\%)$ **through our improved log-aware clone detection tool (LACC).**

## 5.7 Threats to validity

Threats to validity are concerned with the external and internal generalization of our work. External validity discusses how factors outside our dataset can influence our analysis and conclusion, and internal validity concerns with internal assumptions that might impact our analysis. Below, we summarize external and internal threats.

### 5.7.1 External Validity

The external threats to validity reflect on the generalization of our work to other such software projects and programming languages. In this research, we conducted our log statement analysis on three open-source Java projects that are considered log intensive and are well-established projects [78]. We assumed our approach is independent of the underlying programming language that the source code is written in. However, since other software systems, as well as other programming languages, may follow different logging practices, our findings may not accurately extend and generalize to other such systems.

### 5.7.2 Internal Validity

In this work, we initially depend on the code clone detection accuracy. False positive clone detection can impact our analysis. The selection of source code features can also influence the accuracy of log-aware clone detection. In our research, we studied *"method-level"* (in this case Java methods) code snippets which prior research has shown are the most meaningful type of clones for software evolution and maintenance [178]. However, this assumption might put a limitation on the number of clone snippets that we can detect. One approach is to generalize this study to an arbitrary code block, and, therefore, be able to detect more elementary code clone snippets and have better matches on each possible piece of the source code.

## 5.8 Closing Remarks

Software developers insert logging statements in the source code in various places in order to improve software development and diagnosability. Nevertheless, this process is currently mostly manual, and it does lack a unified guideline for the location and content of log statements. To contribute to log statement automation, in this chapter, we presented an experimental study with pertinent ROs on the log statements' placement in open-source Java projects with the application of code clones to detect the location of log statements. Our analysis of three open-source Java systems shows that 98% of code clones are consistent in their logging statements. Our experiments show understanding, detecting, and utilizing the clone snippets in the source code is beneficial for predicting the location of log statements. Hence, we proposed LACC, which achieves a higher accuracy (90%+) of log prediction when compared to the state-of-the-art clone detectors.

As our future work, we look into expanding the log-aware source code features which incorporate log-related behavior and are beneficial for achieving higher accuracy of log statements' placement prediction. We will pursue developing a tool to predict and suggest not only the location, but also the content and severity level of log statements for an unlogged code snippet by searching among its logged clone pairs.

# Chapter 6

# Borrowing from Similar Code: A Deep Learning NLP-Based Approach for Log Statement Automation

**Abstract-** Software developers embed logging statements inside the source code as an imperative duty in modern software development as log files are necessary for tracking down runtime system issues and troubleshooting system management tasks. Prior research has emphasized the importance of logging statements in the operation and debugging of software systems. However, the current logging process is mostly manual, and thus, proper placement and content of logging statements remain as challenges. To overcome these challenges, methods that aim to automate log placement and predict its content, *i.e.*, *'where and what to log'*, are of high interest. Thus, we focus on predicting the *location* (*i.e.*, where) and *description* (*i.e.*, what) for log statements by utilizing source code clones and natural language processing (NLP), as these approaches provide additional context and advantage for log prediction. Specifically, we guide our research with three research questions (RQs): **(RQ1)** how similar code snippets, *i.e.*, code clones, can be leveraged for log statements prediction? **(RQ2)** how the approach can be extended to automate log statements' descriptions? and **(RQ3)** how effective the proposed methods are for log location and description prediction? To pursue our RQs, we perform an experimental study on seven open-source Java projects. We introduce an updated and improved log-aware code-clone detection method to predict the *location* of logging statements (RQ1). Then, we incorporate natural language processing (NLP) and deep learning methods to automate the log statements' description prediction (RQ2). Our analysis shows that our hybrid NLP and code-clone detection approach (*NLP CC'd*) outperforms conventional clone de-

tectors in finding log statement *locations* on average by 15.60% and achieves 40.86% higher performance on BLEU and ROUGE scores for predicting the *description* of logging statements when compared to prior research (RQ3). Our work demonstrates the effectiveness of borrowing context from similar code snippets for automated log *location* and *description* prediction.

**Keywords:**
software systems, software automation, logging statement, logging prediction, source code, code clones, deep learning, NLP

**An earlier version of this chapter is under review in the ACM Journal of Transactions of Software Engineering and Methodology (TOSEM) [124].**

## 6.1 Introduction

Developers embed logging statements into the software's source code to gather feedback on the computer systems' internal state, variables, and runtime behavior as a common practice. While the system is running, the output of logging statements is recorded in log files, which developers and system administrators will review at a later time for various purposes, such as anomaly and problem detection [311, 116], log message clustering [217, 300], system profile building, code quality assessment [279], and code coverage analysis [82]. Additionally, the importance and depth of knowledge available in logs has also flourished the development of commercial log analysis platforms such as Splunk [43] and Elastic Stack [22]. Figure 6.1 shows a log print statement (LPS) that contains a textual part indicating the context of the log, *i.e.*, *description*, a *variable* part, and a log *verbosity level* indicating the importance of the logging statement and how the level represents the state of the program. Verbosity levels for Log4j [15], an Apache logging Library, include: *trace, debug, info, warn, error, and fatal.*

```
log.warn("Cannot find BPService for bpid=" + id);
  level |       description (LSD)          | variable
```

Figure 6.1: A log example with *level*, *description*, and *variable* parts.

Due to the free-form text format of log statements and lack of a general guideline, adding proper logging statements to the source code remains a manual, inconsistent, and often an error-prone task [78, 79]. In addition, in some cases developers forget to even

add a log statement in the first place, *i.e.*, missing log statements [142, 202]. Moreover, because logging inherently introduces development and I/O cost [339, 104, 127], developers often struggle to decide the *number*, *location*, and *description* of logging statements efficiently [166]. Logging insufficiently may cause missing important runtime information that can negatively affect the postmortem dependability analysis [324], and unnecessary logging can consume extra system resources at runtime and impair the system's performance as logging is an I/O intensive task [339, 104, 127].

Motivated with the aforementioned challenges, *i.e., 1) ad hoc and forgetful logging practices, and 2) cost associated with inefficient logging*, prior methods aiming to automate logging ***location*** and predict the **l**og **s**tatements' ***description*** (LSD), *i.e.*, the *'static text'* of logging statement, are high in demand. These approaches generally aim to automate the logging process and predict whether or not a code snippet needs a logging statement by utilizing machine learning techniques to *train* a model on a set of logged code snippets, and then *test* it on a new set of unlogged code [117, 346] (supervised learning). However, prior research has come short in proposing a general solution based on an arbitrary code snippet, as they mostly concentrate on error log statements, such as *exception* handling [346]. Another group of studies [145, 123] have shown the feasibility of predicting the log statements by the use of **similar code snippets** or code clones[1]. Most recently, Gholamian [122] produced a research plan and presented the motivation behind using code clone detection and NLP approaches to automate logging statements for new code snippets by borrowing the logging context from an already-existing similar code. In summary, the idea is that similar code snippets, *i.e.*, clone pairs, follow similar logging patterns.

Accordingly, for introducing a general approach for log prediction and automation in our work, we study the application of similar code snippets and natural language processing (NLP) techniques for suggesting logging statements. In addition, our approach provides additional context to also predict other details of log statements, such as the LSD, which other log prediction approaches are unable to do. Initially, our study reveals that although using similar code snippets facilitates log automation, however, currently available tools for similar code detection (*i.e.*, general-purpose clone detectors, such as Oreo [270]) cannot be used out-of-the-box for log statement automation. Thus, we first show that in order to enable log statement borrowing from similar code snippets, general-purpose code clone detectors require to be updated to understand logging statements. We perform this task by introducing LACCP (**l**og-**a**ware **c**ode **c**lone **p**lus), an improved code clone-based log location predictor **(Finding 1)**. We then enable predicting the log statements' *description* based on borrowing the logging statement from its clone pair and applying NLP and deep

[1]For code clones, we consider a wide range of similar code snippets, *i.e.*, this includes copy-pasted code clones up to semi semantic clones in the twilight zone [270].

learning approaches **(Finding 3)**. Finally, we evaluate the effectiveness of our proposed approaches for log location and description automation with prior work and show that our approach outperforms conventional clone detectors in finding code snippets which require logging statement on average by 15.60% and achieves 40.86% higher performance on BLEU and ROUGE scores for predicting the *description* of logging statements **(Findings 2 & 4)**. In sum, our contributions in this research are as follows:

- We propose an **improved** log-aware clone detection tool (LACCP), which was initially introduced as LACC [123] for log statements' *'location'* prediction, by resolving two of the clone detection shortcomings (§ 6.3).
- We introduce an algorithm to utilize *LACCP* for LSD prediction and introduce a deep-learning NLP-based approach, "*(NLP CC'd)*"[2], to work in collaboration with *LACCP*, and to improve the performance of log statements' description prediction (§ 6.4). We make our data available for both LACCP and NLP CC'd to encourage comparison and further research [12].
- We provide experimentation on several projects and measure Precision, Recall, F-Measure, and Balanced Accuracy, and compare LACCP's performance with general-purpose state-of-the-art clone detectors, Oreo [270] and LACC [123]. In addition, we calculate the BLEU and ROUGE scores for our auto-generated log statements' *descriptions* with considering different sequences of LSD tokens, and compare our performance with the prior work [145] (§ 6.5).
- We present a case study of the application of our tool for log description prediction in the real world and show how *(NLP CC'd)* can facilitate the software development process (§ 6.6).

The rest of this chapter is organized as follows. Section 6.2 explains our motivation and methodology for log prediction with borrowing from similar code, and in Sections 6.3, 6.4, and 6.5, we investigate RQ1, RQ2, and RQ3, respectively. We then provide a case study of the application of our approach in Section 6.6, and discuss the applicability of our approach in Section 6.7. We provide the threats to the validity of our research in Section 6.8 and review related work in Section 6.9. At last, we present our conclusions and future directions in Section 6.10.

[2]The name resembles NLP-aware Code-Clone-based LSD suggestion.

## 6.2 Motivation and Methodology

This section describes the motivation behind utilizing similar code snippets for log prediction followed by our methodology.

### 6.2.1 Motivation

Improving logging quality with automated approaches is a crucial problem in software development as it helps to enhance the overall code quality [279] and makes the system easier to debug [324]. Thus, *'where and what to log'* are the major challenges to tackle when developing tools to help developers with better logging. Prior research [123, 122] shows that code clones resemble similar logging patterns and proposes an approach that utilizes them for log statement prediction by extracting features from method-level code blocks containing a logging statement.

### 6.2.2 Code Clones

Similar code snippets (*i.e.*, code clones) are code snippets that semantically are similar but can be syntactically different [270]. There are four main classes of code clones [261]: Type-1, which is simply copy-pasting a code snippet, Type-2 and Type-3, which are clones that show syntax differences to some extent, and finally Type 4, which represents two code snippets that are semantic clones, *i.e.*, they are syntactically very different but semantically equal [122]. Figure 6.2 shows the recursive versus iterative implementations of the *binary search* (BS). The logging pattern in the original code, $MD_i$ on Line 9 can be learned to suggest logging statements for its clone, $MD_j$, which is missing a logging statement.

Thus, to predict log statements, we define relevant log-aware source code features and employ them for predicting whether a newly composed method code block requires a log statement. We use method-level clones (rationale explained in the following) and apply different categories of source-code features and feed them into a machine learning tool to identify similar code snippets (*i.e.*, clone pairs), which also follow similar logging patterns. Formally speaking, assuming set $CC_{MD_i}$ is the set of all code clones of Method Definition $MD_i$, if $MD_i$ has a log print statement (LPS), then its clones also have LPSs:

$$\exists LPS_i \in MD_i \implies \forall MD_j \in CC_{MD_i}, \exists LPS_j \in MD_j.$$

```
1 // Original code - MD_i
2 int BS_recursive(A[], Key, l, h)
3 {
4  if(l<=h)
5  {
6   mid=(l+h)/2;
7   if(Key==A[mid])
8   {
9    log.info("Found Key %d at Index %d.
     ",Key,mid)
10    return mid;
11   }
12   else if(Key<A[mid])
13    return BS_resursive(A[], Key, l,
     mid-1);
14   else if(Key>A[mid])
15    return BS_resursive(A[], Key, mid,
     h);
16 }
17 return -1 //not found
18 }
```

```
1 // Clone Type 4 - MD_j
2 int BS_iterative(A[], Key, l, h
  )
3 {
4  while (l<=h)
5  {
6   mid=(l+h)/2;
7   if(Key==A[mid])
8    return mid;
9   else if (Key>A[mid])
10    l=mid+1;
11   else
12    h=mid-1;
13 }
14 return -1; //not found
15 }
```

Figure 6.2: Log prediction with similar code snippets, *i.e.*, semantic clones. On the left side, we observe the recursive psuedocode implementation of the binary search ($MD_i$), and on the right the iterative version ($MD_j$). Borrowing from similar code, the logging statement for $MD_j$ can be learned from its clone logging statement on Line 9 of $MD_i$.

### 6.2.3 Why Leveraging Code Clones for Log Prediction?

We observe the benefits of using our approach in utilizing similar code snippets to borrow logging patterns are threefold:

❶ Clone detection methods are already a part of the software maintenance process. Therefore, it is beneficial if we rely on approaches that already exist in the development process. It enables the reuse of stable tools and techniques, saves on development cost, and expedites the process.

❷ Although we acknowledge that in some cases, code clones are the outcome of shallow

copy-pasting which results in log-related anti-patterns (*i.e.*, issues) [202], this, simultaneously, shows the potential of code clones as a starting point for automated log suggestion and improvement. In other words, by automating and enhancing the log statements in the clone pairs, we can expedite the development process and avoid shallow copy-pasting that developers tend to do. Additionally, by automation, we reduce the risk of irregular and *ad hoc* developers' logging practices, *e.g.*, forgetting to log in the first place.

❸ A significant amount of research is conducted towards improving clone detection in identifying semantic clones [121, 320, 303]. Thus, we foresee our approach becomes emboldened as clone detection approaches grow to be more elaborate, and code reuse and context borrowing will further facilitate and expedite the software development process [121].

### 6.2.4 Method-Level Log Prediction Rationale

In our approach, we decide whether a method code block requires a logging statement, *i.e.*, method-level log decisions. Although finding similar code snippets, and subsequently, log statement prediction can be performed in different granularity levels, such as files, classes, methods, and code blocks, however, method-level clones appear to be the most favorable points of re-factoring for all clone types [178, 122]. This approach also includes all of the logging statements which are nested inside more preliminary code blocks within method definitions, *viz.*, logging statements nested inside code blocks, such as *if-else* and *try-catch*. We also hypothesize that the core idea of our research, *i.e.*, context borrowing from similar code snippets, can be extended to an arbitrary code snippet without major changes.

### 6.2.5 Practical Scenario

A practical scenario that showcases the usability of our approach for log statement location and description during the software development cycle is as follows. We consider a possible employment of our research as a recommender tool, which can be integrated as a plugin to code development environments, *i.e.*, IDE software. Alex is a developer working on a large-scale software system, and he has previously developed method $MD_i$ in the code base. At a later time, Sarah, Alex's colleague, is implementing $MD_j$, which does not have a logging statement yet. Our automated log suggestion[3] approach can predict that

[3]We use *'automation'*, *'suggestion'*, and *'prediction'* interchangeably.

if this new code snippet, $MD_j$, requires a logging statement by finding its clone, $MD_i$, in the code base. Then, the tool can suggest Sarah, just in time, to add a logging statement based on the prediction outcome. Although prior work has shown the majority of the clone pairs match in their logging behavior [123], when there is a conflict among clones, we can return a list of suggestions to the developer and in the end, the developer will make the final decision. This approach can likewise recommend the logging statement description by retrieving the text existing in its clone log statement and the description predicted by the NLP model. This situation can be extended to several source code projects, and bring intra- and inter-project logging suggestions. The latter is useful for small projects where there is not a sufficient prior code base, which is commonly referred to as the project's "cold start" phase [134].

### 6.2.6 Research Questions

We guide our study with the following research questions (RQs):

**(RQ1)** How code clones can be used for automated log location prediction?

**(RQ2)** How the available context from clone pairs can be borrowed for log description prediction?

**(RQ3)** How the accuracy of both log location and description prediction can be evaluated and compared with prior work?

For RQ1, we first expose two shortcomings of general-purpose clone detection, and then improve on the performance of clone detection methods to make them more suitable for effective log statement *location* prediction (§ 6.3). For RQ2, we predict the *description* of logging statements in methods without logging statements by searching for the logging descriptions that we can obtain from their clone pairs. Additionally, to enhance the LSD prediction, we apply NLP-based deep learning (DL) methods (*NLP CC'd*) and further improve the LSD prediction when compared to the LSD retrieved from the clone pair (§ 6.4). Finally, we evaluate the performance of both log location and description prediction in RQ3 (§ 6.5).

## 6.3 RQ1: How code clones can be used for automated log location prediction?

### 6.3.1 Motivation and Approach

Prior work has shown similar code snippets have similar logging characteristics [145, 123, 122]. This finding opens up a potential way to automate logging statements' locations [122]. The approach in essence is to find similar code snippets to the code that is currently being developed, and make a logging decision for the new code by observing the logging patterns of its similar code. This automated log suggestion approach can help developers in making logging decisions and improve logging practices.

### 6.3.2 Findings

Although the proposed approach has potential for log automation, during our initial manual scrutiny of detected and undetected similar code snippets, we observe that due to **two shortcomings** that exist in the prior work, we have not been able to gain the full benefit of the log automation through similar code. In particular, we discover that although prior work [123] observed satisfactory prediction scores for the projects under study with LACC and outperforms Oreo [270], it falls short in balancing the Precision and Recall of predictions. As such, we hypothesize that the log-prediction performance of LACC can be improved by recognizing some of the *'undetected (false negative)'* and *'mis-detected (false positive)'* cases of clone detection, which are directly pertinent to the existence or absence of the logging statements. In the following, we assume $(MD_i, MD_j)$ are methods with logging statements that initially detected as clone pairs, and the prime-symbol version ($'$), *e.g.*, $MD_{j'}$, is obtained from the method after removing its logging statement.

#### Shortcoming I (SI) - high rate of undetected clones.

This scenario happens when the logging statement(s) from the method definition in the code base ($MD_i$) contribute to a significant portion of the method body. $(MD_i, MD_j)$ are detected as clone pairs, because both of them have a logging statement and thus their code feature values match. However, in a real-world scenario, when $MD_{j'}$ is being just developed and does not have a logging statement yet, the method's source code metrics for $MD_{j'}$ will be significantly different from $MD_i$, which results in $MD_i$ and $MD_{j'}$ to not be detected as clone pairs *(i.e.*, false negative) [123].

**Shortcoming II (SII) - high rate of mis-detected clones.**

Because log statements do not change the semantics of the source code, we argue that two code snippets should be clones regardless of the existence of their log statements. We notice there are a considerable number of code clones that are not matched as clone pairs after the log statements (including the log statement static text) are removed from both clone pairs, $MD_{i'}$ and $MD_{j'}$, *i.e.*, false positive. In other words, log statements had a critical role in matching the clone pairs, and if log statements are removed from both code snippets, then the code snippets are no longer detected as clone pairs. Considering that log statements are for book-keeping purposes, and do not change the semantics of the program, we reckon this case as a false positive clone detection that only relies on the similarities of log statements rather than the *semantic-effective* lines of the source code. Listing 6.1 illustrates this case as $MD_i$ and $MD_j$ are only matched because of the similarity in their logging statements, which is a book-keeping aspect and does not change the semantics of the code.

**Overcome the Shortcomings**

The observations from **SI** and **SII** confirm that general-purpose clone detection cannot readily be applied for log suggestion, as we are looking to suggest a log statement for a newly-developed code snippet (*i.e.*, without a logging statement) by finding its clone pairs that already have logging statements. As such, with log-aware feature calculation in LACCP, we aim to achieve a higher performance in clone-based log statement automation.

Table 6.1 presents log-related features which are utilized for detecting method clones with a logging statement. These features are in three main categories: *numeric, boolean, and string*. For example, *LWK* represents log related keywords and wrappers, *e.g.*, 'log.info' and 'logger.debug' as string features. The selected features in Table 6.1 enable us to recognize the logging statements and consider them respectively in source code feature calculation. We have surveyed the features used in prior work [275, 270] and experimented with them, *i.e.*, with feature selection and extraction [174], and measured the performance metrics such as *Precision* and *Recall*, and picked the features which help the most with log prediction accuracy.

### 6.3.3 Log-Aware Feature Calculation Illustrative Example

To elaborate further on the inner workings of LACCP, we provide the following example. The idea is to examine how the method-level features from Table 6.1 are calcu-

*Listing 6.1: Wrong clone detection because of log statements.*

```
1  //MD_i, logging statements are commented.
2    protected byte[] createPassword(NMTokenIdentifier identifier)
     {
3      //LOG.debug("creating password for {} for user {} to run on
       NM {}",
4      //    identifier.getApplicationAttemptId(),
5      //     identifier.getApplicationSubmitter(), identifier.
       getNodeId());
6      readLock.lock();
7      try {
8        return createPassword(identifier.getBytes(),
9            currentMasterKey.getSecretKey());
10     } finally {
11       readLock.unlock();
12     }
13   }
14
15 //MD_j, logging statements are commented.
16 protected byte[] retrivePasswordInternal(NMTokenIdentifier
   identifier,
17       MasterKeyData masterKey) {
18     //LOG.debug("retriving password for {} for user {} to run on
        NM {}",
19     //     identifier.getApplicationAttemptId(),
20     //     identifier.getApplicationSubmitter(), identifier.
       getNodeId());
21     //LOG.debug("Response line: " + identifier.getResponseLine()
       );
22     return createPassword(identifier.getBytes(), masterKey.
       getSecretKey());
23   }
```

| Feature | Description | Type |
|---|---|---|
| ELPS | Existence of an LPS | Boolean |
| NTOK | Number of tokens | Numerical |
| NOS | Number of statements | Numerical |
| NEXP | Number of expressions | Numerical |
| LMET | Number of local methods called | Numerical |
| XMET | Number of external methods called | Numerical |
| SLOC | Source lines of code | Numerical |
| LWK | Logging wrappers and keywords | String |

Table 6.1: Method-level log related features.

lated with and without LACCP. In Table 6.1, *SLOC* includes all lines of the source code, such as comments, brackets ({}) for *if-else* blocks, *etc.*; however, *NOS* includes only executable expressions in the method definition. Following examples are based on Line 21: *"LOG.debug("Response line: " + identifier.getResponseLine())"* from Listing 6.1.

**ELPS:** there exists logging statements for this method, therefore, $ELPS_{MD_i} = True$.

**NTOK:** in Listing 6.1, removing this line causes the number of tokens for $MD_{j'}$ (*i.e.*, $NTOK_{MD_{j'}}$) to decrease by 6 when compared to $NTOK_{MD_i}$; tokens are *'LOG'*, *'debug'*, *'Response'*, *'line'*, *'identifier'*, and *'getResponseLine'*.

**SLOC, NOS, and NEXP:** similar to NTOK, $SLOC_{MD_{j'}}$, $NOS_{MD_{j'}}$, and $NEXP_{MD_{j'}}$ values reduce by one as an executable line of $MD_{j'}$ has been removed.

**LMET and XMET:** these values represent the number of local and external method calls. $LMET_{MD_{j'}}$ and $XMET_{MD_{j'}}$ values also decrease as the omitted line makes references to other methods, both internal, *'getResponseline'*, and external, *'debug'*.

**LWK:** we also search and find a comprehensive set of log related keywords, *e.g.*, 'log.info', 'logger.debug', *etc.*, as string features, which come into consideration in LACCP. Table 6.2 summarizes the changes in feature values for $MD_j$ and $MD_{j'}$.

In a real scenario, $MD_i$ is previously developed and is in the code base, and we are looking to automate logging for its clones which are being currently developed without logging statements (*i.e.*, $MD_{j'}$). Thus, the more logging statements exist in method $MD_i$, the more source code features will diverge for $MD_i$ and $MD_{j'}$, and thus it becomes more troublesome for general-purpose clone detectors to detect them as clone pairs. In LACCP's design, for each method $MD_i$ with a logging statement, we calculate the features by rec-

| Feature | Value ($MD_j$) | Value ($MD_{j'}$) |
|---|---|---|
| ELPS | True | Flase |
| NTOK | $NTOK_{MD_j}$ | $NTOK_{MD_j} - 6$ |
| SLOC | $SLOC_{MD_j}$ | $SLOC_{MD_j} - 1$ |
| NOS | $NOS_{MD_j}$ | $NOS_{MD_j} - 1$ |
| NEXP | $NEXP_{MD_j}$ | $NEXP_{MD_j} - 1$ |
| LMET | $LMET_{MD_j}$ | $LMET_{MD_j} - 1$ |
| XMET | $XMET_{MD_j}$ | $XMET_{MD_j} - 1$ |
| LWK | *LOG.debug* | *None* |

Table 6.2: Log-related features comparison with ($MD_j$) and without ($MD_{j'}$) the log statement.

ognizing the logging code first and then exclude its impact on the values of the features in Table 6.1. Feature values are updated such that methods are detected as clone pairs regardless of the presence of logging statements, *i.e.*:

$$Fr(MD_i) \sim_{clone} Fr(MD_j) \implies Fr_{LACCP}(MD_i) \sim Fr(MD_{j'}) \implies Fr_{LACCP}(MD_{i'}) \sim Fr(MD_{j'}).$$

We then add the methods which satisfy the above condition to the set of clone pairs for $MD_i$. In addition to the features in Table 6.1, we also utilize the other features listed in [270] for general clone detection, however, we only perform ***log-aware*** feature calculation on features in Table 6.1. Since log statements do not directly change other feature values, we refer the reader to [270] for further details. An example of features that log statements do not generally have impact on is the number of loops, *i.e.*, *for and while*.

### 6.3.4 Approach Significance

Prior approaches [346, 200] rely on extracting features and training a learning model on logged and unlogged code snippets. Thus, they can predict if a new unlogged code snippet needs a logging statement by mapping its features to the learned ones. Although these methods initially appear similar to our approach in extracting log-aware features from code snippets, *i.e.*, Table 6.1, we believe our approach has an edge over the prior work. Because we also have access to the clone pair of the code under development, *i.e.*, $MD_i$ in $(MD_i, MD_{j'})$, this enables us to obtain and borrow the additional context from $MD_i$ to predict other aspects of log statements, *e.g.*, the LSD, which the prior work is unable

to do. The significance of our approach becomes apparent in LSD automation (§6.4) as we utilize the LSD of the clone pair as a starting point for suggesting the LSD of the new code snippet. Thus, our approach not only complements the prior work in providing logging suggestions for developers as they develop new code snippets, but it also has an edge over them by providing additional context for further prediction of LPS details, such as the LSD and the log's verbosity level. Moreover, prior research has shown [123] that there exists a significant portion of clone pairs of Type 3 and above, *i.e.*, code pairs that are considerably different in syntax or they are semantic pairs. Although later on we evaluate and show the applicability of our approach on a set of limited projects, we envision that our approach would be of a greater significance for a large collection of software, *e.g.*, thousands of projects from GitHub. This way, semantic clone pairs can be found across different projects and used to borrow and predict log statements.

## 6.4 RQ2: how the available context from clone pairs can be borrowed for log description prediction?

### 6.4.1 Motivation

Based on the approach for predicting the location of logging statements with similar code snippets in RQ1 and the additional available context from the clone pairs, *i.e.*, the logging statement description available from the original method, $MD_i$, we hypothesize it is a valuable research effort to explore whether it is also possible to predict the logging statements' *description* automatically. With satisfactory performance, an automated tool that can predict the description of logging statements will be a great aid, as it can expedite the logging process and improve the quality of logging descriptions.

### 6.4.2 NLP for LSD Prediction - Theory

The predictable and repetitive characteristics of common English text, which can be extracted and modeled with statistical natural language processing (NLP) techniques, have been the driving force of various successful tasks, such as speech recognition [59] and machine translation [219]. Prior research [155, 298, 51, 126] has shown that software systems are even more predictable and repetitive than common English, and language models perform better on software engineering tasks than English text; tasks such as code completion [262] and suggestion [66]. Most recently, He *et al.* [145] and Gholamian and

Ward [126] showed that logging descriptions in the source code and log files also follow natural language characteristics. Thus, we introduce a deep learning (DL) natural language model to auto-generate the log statements descriptions. Intuitively, if there is observable repetitiveness in logging descriptions, the trained model should have acceptable prediction performance for new logging statements.

There are two main categories of language models (LMs): 1) statistical LMs which utilize n-gram [9] and Markovian distribution [7] to learn the probability distribution of words, and more recently, 2) deep learning (DL) LMs which have surpassed the statistical LMs in their prediction performance, as they can capture more long-range token dependencies [307, 97]. Thus, in this research, we utilize deep learning LMs. Once LMs are trained on sequences of tokens or n-grams (*e.g.*, words), they can assign scores and predict the probability of new sequences of words. Considering a sequence of tokens in a text (in our case, logging statement description, LSD), $S = a_1, a_2, ..., a_N$, the LM statistically estimates how likely a token is to follow the preceding tokens. Thus, the probability of the sequence is estimated based on the product of a series of conditional probabilities [155]:

$$P_\theta(S) = P_\theta(a_1)P_\theta(a_2|a_1)P_\theta(a_3|a_1a_2)....P_\theta(a_N|a_1...a_{N-1})$$

which is equal to:

$$P_\theta(S) = P_\theta(a_1).\prod_{t=2}^{N} P_\theta(a_t|a_{t-1}, a_{t-2}, ..., a_1), \tag{6.1}$$

where $a_1$ to $a_N$ are tokens of the sequence S and the distribution of $\theta$ is estimated from the training set. Given a sequence of log description tokens $a_1, ..., a_t$, we seek to predict the next M tokens $a_{t+1}, ..., a_{t+M}$ that maximize Equation 8.2 [66]:

$$P_\theta(S) = \underset{a_{t+1},...,a_{t+M}}{arg\,max}\ P_\theta(a_1, ..., a_i, a_{i+1}, ..., a_{i+M}) \tag{6.2}$$

As such, an LSTM implementation of the LM to maximize the probability of observing token $a_t$ in Equation 6.2 at time step t is:

$$P_\theta(a_t|a_{t-1}, ..., a_1) = \frac{exp(\upsilon_{a_t}^T h_t + b_{a_t})}{\sum_{a_{t'}} exp(\upsilon_{a_{t'}}^T h_t + b_{a_{t'}})} \tag{6.3}$$

where $h_t$ is the output of the hidden state vector at time t, $\upsilon_{a_t}^T$ is a parameter vector associated with token $a_t$ in the vocabulary and $b_{a_t}$ is a constant value. Intuitively, in Equation 6.3, $\upsilon_{a_t}^T h_t + b_{a_t}$ is a function that shows how much the model favors in observing $a_t$ after the sequence of $a_{t-1}, ..., a_1$, and the *exp()* function assures the values are always positive. The summation in the denominator calculates the probability values of each token over all tokens out of the maximum probability value of 1.

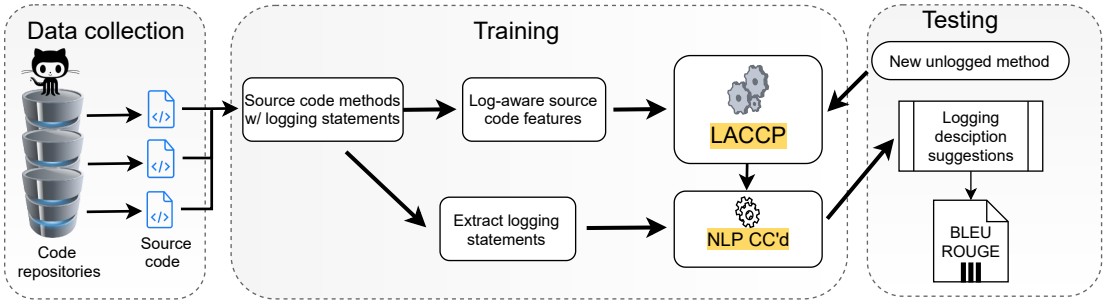



Figure 6.3: The toolchain for log statement description prediction. The approach shows how both LACCP and NLP CC'd collaborate for LSD prediction.

### 6.4.3 Methodology

We base our method on the assumption that clone pairs tend to have similar logging statements' descriptions. This assumption comes from the observations in predicting log statements for clone pairs. As logging descriptions explain the source code surrounding them, it is intuitive for similar code snippets to have comparable logging descriptions. Based on this assumption, we propose a deep learning method that borrows the LSD from similar code snippets and leverages NLP approaches (NLP CC'd). In particular, to generate the LSD for a logging statement in $MD_j$, we extract its corresponding code snippet (*i.e.*, the method without the log statement, $MD_{j'}$) and leverage LACCP to locate its clone pairs. Laterally, the NLP model is trained on the logging descriptions available in the training set for each project. To ensure the training and testing sets are mutually exclusive, for all of the clone pairs of $(MD_i, MD_j)$, the LSDs of $MD_i$s and $MD_j$s create the training and testing data, respectively. During the testing, the retrieved logging description from the clone pair ($LSD_{MD_i}$) is served as a starting input point for the NLP model to propose a set of description suggestions. Then, we evaluate the similarity between the NLP-generated descriptions with the LSD provided by the developers ($LSD_{MD_j}$) (*ground truth*). Our methodology resembles the scenario that while the developer is creating a new snippet of the source code, we look for its similar code snippets with LACCP, and in case a clone is found ($MD_i$), we work further to provide predictions on the description of the logging statement by generating suggestions from the available clone's LSD ($LSD_{MD_i}$) combined with the collective knowledge of LSDs available in the code base.

### 6.4.4 Toolchain

Figure 6.3 presents our toolchain for log statement description prediction. In the **data collection** phase, we select open-source Java projects from their Git repositories, based on factors of interest such as age and size of the project (in source lines of code), popularity (being well-established), stability, and logging index of the projecst [78]. Listed in Table 6.3, we select seven Apache Java projects. Next, commencing in the **training** phase, we extract method definitions (MDs) containing logging statements by applying JavaParser [297]. Initially, we parse the source code to obtain the abstract syntax tree (AST), which is the hierarchical representation of the code. We use the AST to access Java method definitions with logging statements. We then extract method-level code features to perform log-aware clone detection (LACCP) on the extracted method definitions and leverage LACCP to find clone pairs with logging statements. Next, for each detected clone pair, we use the descriptions obtained from $MD_i$s to train the NLP model. Finally, in the **testing** phase, we use the $MD_{j'}$s as test-case inputs to LACCP. The NLP model, upon the clone pair detection of $(MD_i, MD_{j'})$, receives the LSD from $MD_i$ and suggests descriptions with the highest probability for $MD_{j'}$. Then, we compare the NLP-generated LSDs with the logging statement originally placed by developers in $MD_j$, and calculate the BLEU and ROUGE scores.

### 6.4.5 Implementation

The recent advancements in deep learning (DL) approaches have provided a new avenue for log analysis and automation. DL models used in prior logging research [107, 337, 200, 203] have enabled more accurate and elaborate analysis of logs when compared to prior approaches. For our NLP approach, we utilize Long Short-Term Memory (LSTM) [158] deep learning models which are recurrent neural networks (RNNs) capable of capturing long-term dependencies in a sequence of tokens through their internal memory. This feature makes them suitable for LSD prediction in our research, as we are pursuing to predict a sequence of words for the LSDs. Figure 6.4 shows the overall layout of our model, which has an input layer, two hidden layers, a dense layer with Rectified Linear Unit (relu) [235, 260] as the activation function ($relu(x) = max(0, x)$), and an output layer with *softmax* activation. The layers are sized as: input layer is the *'vocabulary size'*, the LSTM layers 1 and 2 are *'500 cells'*, the dense layer is *'250 cells'*, and the output layer has the same size as the input layer. During the training phase, in the first layer of the model, we map the LSD sequences to vectors of integers by leveraging an *'embedding layer'*. The embedding layer infers the relationships among tokens in the LSD input sequences, and

outputs a set of lower-dimension vectors, called word embeddings [227]. The embedded vectors then pass through two layers of LSTM and allow the model to learn the relationship between the sequence of words in the LSD and assign probabilities. Followed by LSTM layers 1 and 2, the dense layer is placed with *relu* activation function. Finally, the output layer produces *softmax* probabilities for each next token prediction in the suggested LSD.

For the DL implementation, we use Python's Keras library [36]. The *relu* activation function allows achieving a non-linear transformation of data, which results in learning more sophisticated patterns and relationships among log statement description tokens. We utilize a *softmax* activation function [70, 312] on the output layer such that the network can learn and output probability distribution over possible next tokens in the sequence of words within the LSD. The *softmax* layer converts the output of the dense layer to a probability distribution of tokens in the log vocabulary. This ensures that the LSTM outputs are all in the range of [0,1], and their summation is equal to one in every prediction [131]. We also apply 10% dropout on the hidden layers as a common practice [289, 248]. With this measure, the output of the 10% of the hidden layer nodes, which are randomly selected during each iteration of the model training, is ignored. This process ensures that the model is generalizable and avoids its overfitting to the training data [289]. We train the model for 200 epochs and set the batch size to 64. During the testing, from the outputted LSDs of the DL model, we pick the highest (NLP-1) or top-3 (NLP-3) softmax probabilities and provide them as suggestions. We have been partially deliberate in the selection of the hyperparameters, and as an avenue for future work, different layouts or hyperparameters' setup, such as more memory cells or deeper layers of LSTM network, may achieve a better performance.

### 6.4.6 LSD Prediction Algorithm and Steps

We follow the steps outlined in Algorithm 2 for LSD suggestion for a method based on the LSD borrowed from its clone pair. After collecting the source code projects from their git repositories, we then parse the source code to its abstract syntax tree (AST) [1], and search for methods with logging statements (Lines 1-2). We define $CC_{MD_i}$ as the set of all clone pairs of Method $MD_i$, *i.e.*,: $CC_{MD_i}(pairs) = \{(MD_i, MD_j) | \forall MD_j \in CC_{MD_i}\}$. On Line 2, after extracting all method definitions with at least one LPS from the AST, we find all clone pairs for each method definition $MD_i$, by applying *LACCP*, in the *for-loop* on Lines 4-9. After finding clones of $MD_i$ and creating $(MD_i, MD_j)$ pairs on Line 6, we add $MD_i$s to the training and $MD_{j'}$s to the testing sets on Lines 8-9 for NLP CC'd to use later for LSD prediction. We also add the pair to the sets of $nlp_{training}$ and $nlp_{testing}$ on Lines 8 and 9. Before training the LSTM model on the retrieved LSD data, we perform **pre-processing** on the LSDs (Line 12) such as tokenization, mapping punctuation to the

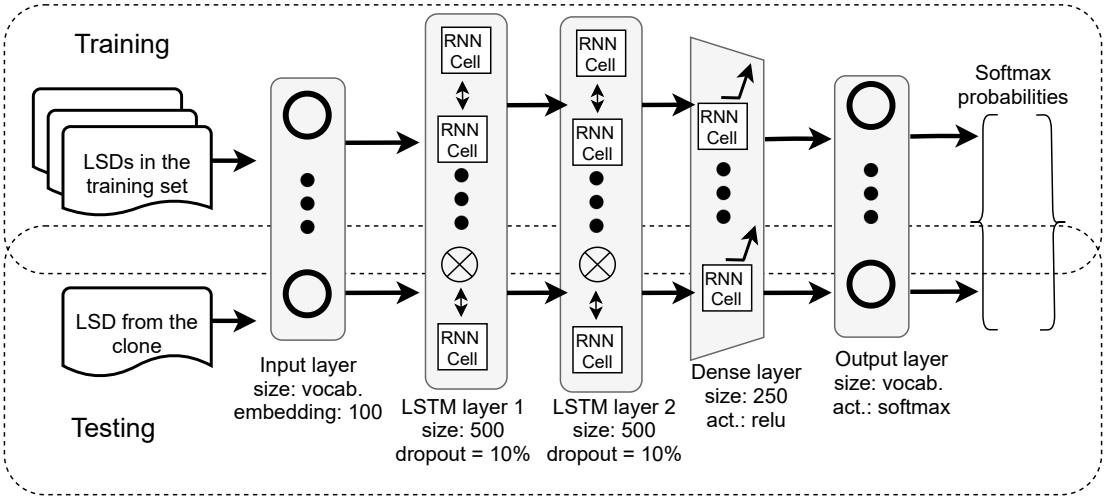



Figure 6.4: The figure shows the inside of NLP CC'd, our deep learning long short-term memory (LSTM) model for log description prediction.

vocabulary space, cleaning of special characters (*e.g.*, Unicode) and removing the LSD parts related to the dynamic variables. On Line 13, we train the NLP model by using the logging descriptions collected in the training set. Then, in the *for-loop* on Lines 14-18, for predicting the LSD for $MD_{j'}$, we feed in the logging description from $MD_i$ to the NLP model, and the trained model returns LSD suggestions.

## 6.5 RQ3: how the accuracy of both log location and description prediction can be evaluated and compared with prior work?

In this section, we evaluate the performance of LACCP (**RQ3.I**) and NLP CC'd (**RQ3.II**).

### 6.5.1 Systems Under Study

Based on the research on open-source projects by Chen and Jiang [78] and He *et al.* [145], we choose seven open-source Java projects. These projects are well-logged, stable, and well-used in the software engineering community, and this selection also enables us to

**Algo. 2: Log Location & Description Predictor (LACCP & NLP CC'd)**

**Input:** Java source code repositories
**Output:** $BLEU_{list}$ and $ROUGE_{list}$ scores

$sourceCode_{AST} \leftarrow Parse(sourceCode)$
$Methods_{w/LPS} \leftarrow extract_{MD}(sourceCode_{AST}, exist(LPS))$;
$nlp_{training} \leftarrow \{\}$; $nlp_{testing} \leftarrow \{\}$;
**for** $(\forall MD_i \in Methods_{w/LPS})$ **do**
    $CC_{MD_i} \leftarrow \texttt{findClones}_{LACCP}(MD_i)$
    $CC\_pairs(MD_i, MD_j) \leftarrow createPairs(CC_{MD_i})$
    **for** $(\forall (MD_i, MD_j) \in CC\_pairs(MD_i, MD_j))$ **do**
        $nlp_{training} \leftarrow MD_i \bigcup nlp_{training}$;
        $nlp_{testing} \leftarrow MD_{j'} \bigcup nlp_{testing}$;
$BLEU_{list} \leftarrow \{\}$;
$ROUGE_{list} \leftarrow \{\}$;
$nlp_{training} \leftarrow preProcess(nlp_{training})$;
$nlpModel \leftarrow train(\forall MD_i \in nlp_{training})$;
**for** $(\forall (MD_i, MD_{j'}) | MD_{j'} \in nlp_{testing})$ **do**
    $nlpLSD \leftarrow nlpPredic(LSP(MD_i))$;
    **for** $\forall LSD_i \in nlpLSD$ **do**
        $BLEU_{list} \leftarrow calcBLEUScores(LSD_i, LPS(MD_j))$;
        $ROUGE_{list} \leftarrow calcROUGEScores(LSD_i, LPS(MD_j))$;
**return** $BLEU_{list}, ROUGE_{list}$;

compare our research with prior work, accordingly. Table 6.3 summarizes the line number of source code (LOC), the number of logging statements (LPS) in each project, and the log density in a thousand lines of code (KLOC). All of the selected projects use Apache Log4j library [15] as the logging statements' wrapper function, which includes six log verbosity levels: *fatal, error, warn, info, debug,* and *trace*. We observed that although all these projects are from the Apache umbrella, they are from different domains, are developed by different teams, and incorporate different logging practices.

| Project (abrv.) | Description | # LOC | # LPS | # of LPS per KLOC |
|---|---|---|---|---|
| Hadoop (HD) | Distributed Computing | 2.10M | 16,202 | 7.72 |
| Zookeeper (ZK) | Configuration management | 94,434 | 1,885 | 19.96 |
| CloudStack (CS) | Cloud deployment | 739,732 | 12,237 | 16.54 |
| HBase (HB) | Distributed database | 949,310 | 9,264 | 9.76 |
| Hive (HI) | Data warehouse | 1.84M | 10,640 | 5.78 |
| Camel (CL) | Integration platform | 2.2M | 9,682 | 4.40 |
| ActiveMQ (MQ) | Message broker | 464,632 | 6,442 | 13.86 |

Table 6.3: The table lists the details for the studied project. The projects are well-established software from different application domains. The table also lists the number of lines of code (LOC), number of log print statements (LPS), and number of log statements per thousand lines of code (KLOC).

### 6.5.2 RQ3.I: LACCP Evaluation

**Evaluation Metrics.**

For evaluating the performance of the logging statement prediction, we utilize Precision, Recall, F-Measure, and Balanced Accuracy.

- *Precision* is the ratio of the correctly predicted log statements ($t_p$) to the total number of predictions ($t_p + f_p$), $Precision = \frac{t_p}{t_p+f_p}$.
- *Recall* is the ratio of $t_p$ over the total number of log statement instances detected and undetected ($t_p + f_n$), $Recall = \frac{t_p}{t_p+f_n}$.
- In order to confirm the balance between the *Precision* and *Recall* values, we also calculate *F-Measure* which is the harmonic average of the *Precision* and *Recall*, $F-Measure = 2 \times \left(\frac{Recall \times Precision}{Recall+Precision}\right)$.
- In addition, to ensure our results are impartial towards the imbalanced datasets [346], we also measure *Balanced Accuracy (BA)*, which is the average of logged and unlogged methods that are correctly predicted, $BA = \frac{1}{2} \times (\frac{t_p}{t_p+f_n} + \frac{t_n}{t_n+f_p})$.

For method-level clone detection in our studied systems, we use the same pre-trained learning model provided in [270] as our baseline. To ensure variability in the dataset, the

machine learning engine of the clone detector is trained on 50K randomly-selected Java projects from Github [270]. Randomly, 80% of each project is selected for training and the remaining 20% for testing. Additionally, one million pairs from the training set of 50K projects are kept separately for validation to avoid bias and overfitting [75]. To ensure a fair comparison, all of the approaches use the same trained model for clone detection, and theretofore, the advantage of LACCP becomes visible from its log-aware source code feature selection and calculation.

**Experiments and Results.**

Our goal in this experiment is to show LACCP's performance in predicting the log locations for clone pairs compared to LACC and Oreo. For experiment design, we first extract all the methods with logging statements and then remove their logging statements. Next, we check which pairs are detected as pairs after their logging statemented are removed, *i.e.*, $(MD_{i'}, MD_{j'})$, which serves as actual true positive cases, $t_p$, and which method are not detected as clone pairs, *i.e.*, $t_n$. This process ensures that the methods are detected as clones regardless of their logging statements, which are considered as *ground truth* for our comparison. In this case that there are no logging statements in the methods, all of the approaches (Oreo, LACC, and LACCP) will have similar performance in clone detection as log-aware feature calculation will have an effect on the feature values only if logging statements are present in the code snippet. Later, we run Oreo, LACC, and LACCP on the collected pairs as $(MD_i, MD_{j'})$ and evaluate their performance compared to the *ground truth*. The experimented scenario resembles the situation when the developer is composing a new source code snippet without a logging statement $(MD_{j'})$, and we look in the code base to find clones with logging statements $(MD_i)$ and provide logging suggestions to the developer.

Table 6.4 shows *Precision*, *Recall*, *F-Measure*, and *BA* for the three approaches. For log statement prediction for seven different projects, the measured values for *BA* are in the range of $(78\%, 92\%)$ for Oreo, $(81\%, 97\%)$ for LACC, and $(95\%, 100\%)$ for LACCP. For example, for Hadoop project, with the number of *ground truth* positive cases of 879, and *ground truth* negative cases of 235, after removing logging statements for $MD_j$s, Oreo had 16 false positives $(f_p)$, a significant number (319) of false negatives $(f_n)$, and 560 true positives $(t_p)$, achieving 'imbalanced' Precision and Recall of 97.22% and 63.71%, respectively. In comparison, for LACCP: $(f_p = 18)$, $(f_n = 17)$, and $(t_p = 862)$, yielding Precision and Recall of 97.95% and 98.07%. Although LACC improves on Oreo by partially reducing the number of false negatives, it still suffers from a high rate of false negatives, which hurts its Recall scores when compared to LACCP. Overall, considering the average of BA values,

LACCP brings 15.60% and 5.93% improvement over Oreo and LACC, respectively, across the experimented projects. The higher accuracy that LACCP brings enables us to provide more accurate clone-based log statement suggestions.

**Qualitative Comparison.**

Intuitively, because LACCP performs log-aware feature selection and calculation, it understands the log feature changes while processing the source code, and it achieves a higher log-aware clone prediction performance compared to the general-purpose clone detector, Oreo [270]. In addition, LACCP, in contrast to LACC that only addresses **SI** and still suffers from a significant number of mis-detected and undetected clones, seeks to simultaneously address both **SI** and **SII** and reaches balanced Precision and Recall scores, and therefore, higher F-Measure and Balance Accuracy (BA) values compared to LACC.

Orthogonal to our research, prior efforts, such as [346], [166], and [200], have proposed learning approaches for logging statements' *location* prediction, *i.e.*, *where to log*. The approaches in [346], [166] are focused on error logging statements (ELS), *e.g.*, log statements in *catch clauses*, and are implemented and evaluated on C# projects. Li *et al.* [200] provide log location suggestions by classifying the logged locations into six code-block categories. Different from these prior works, our approach does not distinguish between error and normal logging statements, is evaluated on open-source Java projects, and leverages logging statement suggestions at method-level by observing logging patterns in similar code snippets, *i.e.*, clone pairs. In addition, none of the aforementioned studies have tackled the automation of the log statements descriptions, which we have also proposed in this research, and will evaluate in the following.

**Finding 1.** *Augmenting general-purpose clone detection approaches with log-aware features is necessary and beneficial for log statement automation, as we aim to predict a logging statement for a code snippet that is unlogged, by looking for its similar clones, which are logged.*

**Finding 2.** *Our results show that LACCP outperforms Oreo and LACC in Balanced Accuracy values by 15.60% and 5.93%, respectively.*

| Proj.\Tool | Oreo [270] (X) | | | | | | | | LACC [123] (Y) | | | | | | | | LACCP (Z) | | | | | | | | Improvement % | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *tp* | *tn* | *fp* | *fn* | *P* | *R* | *F* | *BA* | *tp* | *tn* | *fp* | *fn* | *P* | *R* | *F* | *BA* | *tp* | *tn* | *fp* | *fn* | *P* | *R* | *F* | *BA* | Z(over)X | Z(over)Y |
| HD | 560 | 219 | 16 | 319 | 97.22 | 63.71 | 78.98 | 78.45 | 825 | 218 | 17 | 54 | 97.98 | 93.86 | 95.87 | 93.31 | 862 | 217 | 18 | 17 | 97.95 | 98.07 | 98.01 | 95.20 | 21.35 | 2.03 |
| ZK | 47 | 33 | 0 | 17 | 100 | 73.44 | 84.68 | 86.72 | 62 | 27 | 6 | 2 | 91.18 | 96.86 | 93.94 | 89.35 | 61 | 32 | 1 | 3 | 98.39 | 95.31 | 96.83 | 96.14 | 10.86 | 7.60 |
| CS | 430 | 222 | 22 | 184 | 95.13 | 70.03 | 80.68 | 80.51 | 441 | 223 | 21 | 173 | 95.45 | 71.82 | 81.97 | 81.61 | 610 | 239 | 5 | 4 | 99.19 | 99.35 | 99.27 | 98.65 | 22.53 | 20.88 |
| HB | 377 | 182 | 13 | 115 | 96.67 | 76.62 | 85.49 | 84.98 | 479 | 190 | 5 | 13 | 98.97 | 97.36 | 98.16 | 97.04 | 492 | 195 | 0 | 0 | 100 | 100 | 100 | 100 | 17.67 | 3.05 |
| HI | 586 | 99 | 6 | 66 | 98.99 | 89.88 | 94.21 | 92.08 | 643 | 100 | 5 | 9 | 99.22 | 98.62 | 98.92 | 96.93 | 648 | 100 | 5 | 4 | 99.23 | 99.39 | 99.31 | 97.31 | 5.68 | 0.39 |
| CL | 633 | 270 | 17 | 164 | 97.38 | 79.42 | 87.49 | 86.75 | 746 | 285 | 2 | 51 | 99.73 | 99.73 | 96.57 | 96.45 | 797 | 285 | 2 | 0 | 99.75 | 100 | 99.87 | 99.65 | 14.87 | 3.32 |
| MQ | 338 | 207 | 4 | 131 | 98.83 | 72.07 | 83.35 | 85.09 | 423 | 210 | 1 | 46 | 99.76 | 90.19 | 94.74 | 94.86 | 463 | 209 | 2 | 6 | 99.57 | 98.72 | 99.14 | 98.89 | 16.22 | 4.25 |
| **Total/ Avg** | 2971 | 1232 | 78 | 996 | 97.75 | 75.02 | 84.98 | 84.94 | 3619 | 1253 | 57 | 348 | 97.47 | 92.63 | 94.31 | 92.79 | 3933 | 1273 | 33 | 34 | 99.15 | 98.69 | 98.92 | 97.98 | 15.60 | 5.93 |

Table 6.4: The table shows the value of $t_p$, $t_n$, $f_p$, and $f_n$ for the three approaches. We also show Precision (P), Recall (R), F-Measure (F), and BA for the three methods of log prediction. The general trend on how the methods perform is observable on F-Measure, and BA metrics, as the values increase, *i.e.*, Oreo < LACC < LACCP.

### 6.5.3 RQ3.II: LSD Evaluation

In this section, we measure the performance of (NLP CC'd) for logging statements' *description* prediction. If we can achieve a satisfactory performance, an automated log description predictor that can suggest the description of logging statements will be of great help, as it can significantly expedite the software development process and improve the quality of logging descriptions. To measure the accuracy of our method in suggesting the log description, we utilize BLEU [247] and ROUGE [206] scores. The rationale behind using these scores is that they are well-established for validating the usefulness of an auto-generated text [58, 268, 206, 344, 130]. In particular, prior software engineering and machine learning research have used these scores for tasks such as comment and code suggestion [50] and description prediction [145]. In addition, they are intuitively equivalent to *Precision* and *Recall* for evaluating auto-generated text.

#### BLEU Score

BLEU, or the *Bilingual Evaluation Understudy*, is a score for comparing a candidate text to one or more reference texts. BLEU score is used to evaluate text generated for a series of natural language processing tasks [58, 215]. BLEU score can measure the similarity between a candidate and a reference sentence. In our experiments, we regard the logging description generated by finding the code clone pair snippet and then predicted by the NLP model as the candidate description, while we refer to the original logging description written by the developer as the reference description. BLEU measures how many n-grams (*i.e.*, tokens) in the candidate logging description appear in the reference, which makes it comparable to *'Precision'*. BLEU is evaluated as:

$$BLEU = BP \times exp\left(\sum_{n=1}^{N} w_n \times \log p_n\right) \tag{6.4}$$

where BP is a *'brevity penalty'* that penalizes if the length of a candidate (in number of tokens) is shorter than the reference:

$$BP = \begin{cases} 1 & \text{if c} > \text{r} \\ e^{1-\frac{r}{c}} & \text{if c} \leq \text{r} \end{cases}$$

where $r$ is the length of the reference, and $c$ is the length of the candidate. In Formula 6.4, N is the maximum number of n-grams used in the experiment; $p_n$ is the modified n-gram Precision, which is the ratio of the number of tokens from the candidate logging description

which occur in the reference description to the total number of token in the candidate; and $w_n$ is the weight of each $p_n$. For example, BLEU-1 means the BLEU score considering only the 1-grams in the calculation, where $w_1 = 1$ and $w_2 = w_3 = w_4 = ... = 0$.

From the definition of BLEU, we know that the higher the BLEU, the better the logging statement description prediction performance. The range of BLEU is [0, 1], or as a percentage value (*i.e.*, [0, 100]). Thus, if the candidate logging description does not contain any of the reference's n-grams, then the BLEU score is 0. On the contrary, if all of the candidate tokens appear in the reference, the BLEU score is 100. An additional enhancement to the BLEU score is to calculate cumulative scores as it gives a better sense of the sentence level structure similarity between the candidate and reference descriptions. Cumulative BLEU scores refer to the calculation of individual n-gram scores at all orders from 1 to N and weighting them by calculating the weighted geometric mean. The cumulative and individual 1-gram BLEU use the same weights, *e.g.*, (1, 0, 0, 0). The 2-gram weights assign a 50% to each of 1-gram and 2-gram (*e.g.*, (0.5, 0.5, 0, 0)), and the 3-gram weights are 33% for each of the 1, 2 and 3-gram scores (0.33, 0.33, 0.33, 0).

We provide two sets of BLEU scores: 1) the BLEU scores generated by simply borrowing the LSD from the clone pair as it is (*i.e.*, the LSD of $MD_i$ as a candidate, and the LSD of $MD_j$ as reference), **No-NLP**, and 2) the BLEU scores for the LSD predicted by the NLP CC'd model by considering only one previous token of the input LSD string, LSD($MD_i$), for predicting the next token, *i.e.*, **NLP-1**. The NLP-1 is the output of the LSTM model, and the original LSD of $MD_j$ is considered as the reference for score evaluation. The **No-NLP (X)** and **NLP-1 (Y)** columns in Table 6.5 outline the cumulative BLEU scores for No-NLP and NLP-1. The BLEU-1 scores for all of the evaluated projects are higher than 47%. The highest BLEU-1 score belongs to Hive, 92.50%, which means that 92.50% of the tokens in the generated logging description of the candidate can be found in the ground truth, *i.e.*, the logging description placed in by the developer. This observation implies that in most cases for this project (Hive), developers have reused the logging descriptions from the existing log statements with minor modifications. The NLP-1 model improves the performance of the LSD prediction across all projects.

### ROUGE Score

ROUGE [206] stands for *Recall-Oriented Understudy for Gisting Evaluation*. It includes measures to automatically determine the quality of an auto-generated description by comparing it to other (ideal) descriptions created by humans, *i.e.*, developers in our case. Formally, ROUGE-N is an n-gram Recall between a candidate description and a reference

description. ROUGE-N is computed as follows:

$$ROUGE-N = \frac{\sum_{gram_N \in Ref} Count_{match}(gram_N)}{\sum_{gram_N \in Ref} Count(gram_N)}$$

where $N$ represents the number of overlapping grams that have to match in reference and candidate descriptions; $Ref$ is the reference description; $count_{match}$ $(gram_N)$ is the maximum number of n-grams co-occurring in the candidate and the reference; and $count(gram_N)$ is the number of n-grams in the reference. ROUGE is similar to *'Recall'*, which measures how many n-grams in the reference appear in the candidate logging statement. For example, ROUGE-2 explains the overlap of 2-grams between the candidate and reference descriptions. ROUGE-L is a statistic calculated based on the Longest Common Subsequence (LCS). ROUGE-L takes into account the sentence level structure and similarity, and thus, identifies the longest co-occurring sequence of continuous n-grams. Analogous to BLEU, the range of ROUGE is [0, 1], with 1 being the perfect score, *i.e.*, the candidate description contains all of the reference's n-grams. Similar to BLEU, we provide two sets of ROUGE scores: 1) **No-NLP**, and 2) **NLP-1**. Table 6.5 compares ROUGE scores for No-NLP versus NLP-1. Similar to BLEU, the NLP model suggests LSDs with higher ROUGE scores in all cases when compared to the LSD obtained from the clone pair.

### NLP Prediction Example

To illustrate how the NLP model can improve BLEU and ROUGE scores, we provide the following real example. In the **training set** for CloudStack we have the following LSDs: *"**successfully deleted** condition"* (1x), *"elastistor volume **successfully deleted**"* (3x), and *"**successfully created** floating ip"* (1x). During **testing**, the retrieved LSD from the clone pair, $MD_i$, is *"successfully created floating ip"*, and the original LSD that we are aiming to predict, *i.e.*, the reference LSD of $MD_j$, is *"**successfully deleted** floating ip"*. Because in the training set the token *"deleted"* appears four times (*i.e.*, 1x+3x=4x) right after the token *"successfully"*, whereas *"created"* appears only once (1x) immediately after *"successfully"*, the NLP model assigns a higher probability for observing *"deleted"* after *"successfully"*. Therefore, the NLP model allows us to see highly probable next tokens that appear in the training set as a whole that might not necessarily happen in the LSD retrieved from the clone pair.

### LSD Sequence Prediction - NLP-1 vs. NLP-3

Because LSDs are a sequence of natural language tokens [145, 126], we hypothesize that considering additional prior tokens for predicting the next token will achieve higher performance. As such, in NLP-3, in contrast to NLP-1 that we consider only the one prior token, we take into account the sequence of three prior tokens in predicting the next token. Additionally, because the outputs of the NLP CC'd model are *softmax* probabilities (Figure 6.4), we report the top-3 probabilities for recommending the next token and then confirm which one achieves higher BLEU and ROUGE scores. This approach resembles the scenario in which a list of high probable next tokens is suggested to the developer while composing the LSD. In Table 6.5, **NLP-3 (Z)** illustrates the improvement that NLP-3 brings compared to NLP-1. The rationale behind choosing three prior tokens and not considering longer sequences for LSD prediction is that LSDs are naturally shorter than English text sentences and considering more than four continuous tokens, *i.e.*, three prior tokens and the one token under prediction, would result in a minimal gain or even might cause inaccuracies for shorter than four-token LSDs. In sum, our LSTM model generates candidate LSDs with higher BLEU and ROUGE scores in all cases when compared to the LSD borrowed from the clone pair, and additionally, we enhance the NLP CC'd performance by leveraging a sequence of tokens for LSD prediction. Our experiments signify the benefits of the collaboration of LACCP and NLP CC'd for LSD prediction.

### Results Review and Comparison

In Table 6.5, the BLEU and ROUGE scores gradually decrease as the n-grams grow longer. For example, for **NLP-1**, BLEU-1 for Hadoop is 58.87, while the corresponding BLEU-4 is 33.27. This observation is expected because the BLEU-4 score considers the match of 4 consecutive tokens (*i.e.*, 4-grams) versus BLEU-1, which only considers matching 1-grams.

To provide an intuitive understanding of how good our BLEU and ROUGE scores are, we compare as follows: the BLEU-4 scores of our NLP method outperform prior efforts in [213] and [145]. From a practitioner perspective, the satisfactory BLEU-4 scores reported in the state-of-the-art code summarization paper [213], ranges from 6.4% to 34.3%, which are lower than our reported values. The authors in [213] showed that with their achieved BLEU scores, their auto-generated code summaries are both *fluent* and *informative* for the human reader.

For a direct comparison, we use He *et al.*'s sample data available in [33]. The scores only exist for five projects of our interest (Hadoop, CloudStack, HBase, Hive, and Camel) [145], and their approach includes pairs of *'(code, log)'*, with *'code'* indicating the ten lines of code

| Prj. | No-NLP % (X) | | | | | | | | NLP-1 % (Y) | | | | | | | | NLP-3 % (Z) | | | | Improv. $Z_{(over)}X$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | B-1 | B-2 | B-3 | B-4 | R-1 | R-2 | R-3 | R-L | B-1 | B-2 | B-3 | B-4 | R-1 | R-2 | R-3 | R-L | B-1 | B-4 | R-1 | R-L | B-4 | R-L |
| HD | 58.42 | 48.09 | 39.43 | 32.91 | 59.22 | 36.00 | 20.13 | 59.07 | 58.87 | 48.63 | 40.18 | 33.27 | 53.77 | 32.89 | 59.40 | 58.92 | 60.22 | 34.97 | 62.39 | 62.18 | 6.26 | 5.26 |
| ZK | 64.15 | 56.85 | 47.02 | 41.59 | 63.45 | 46.31 | 25.64 | 63.45 | 65.15 | 58.26 | 48.53 | 43.35 | 66.41 | 50.00 | 30.44 | 66.41 | 66.05 | 43.46 | 68.53 | 68.53 | 4.50 | 8.01 |
| CS | 47.45 | 39.50 | 33.98 | 29.42 | 49.16 | 33.30 | 22.23 | 48.68 | 47.87 | 39.89 | 34.38 | 29.79 | 50.45 | 34.50 | 23.19 | 49.97 | 49.92 | 32.26 | 53.53 | 53.02 | 9.65 | 8.92 |
| HB | 82.67 | 77.37 | 71.78 | 56.80 | 83.15 | 71.68 | 60.86 | 83.06 | 83.18 | 78.02 | 72.40 | 47.38 | 84.32 | 73.37 | 62.15 | 84.23 | 83.67 | 58.38 | 84.64 | 84.53 | 2.78 | 1.77 |
| HI | 92.50 | 91.07 | 89.62 | 70.57 | 92.61 | 88.21 | 85.36 | 92.57 | 92.55 | 91.10 | 89.64 | 70.58 | 92.65 | 88.30 | 85.41 | 92.61 | 92.64 | 70.61 | 92.73 | 92.69 | 0.06 | 0.13 |
| CL | 66.36 | 55.65 | 49.15 | 41.17 | 65.34 | 42.48 | 25.24 | 64.65 | 66.51 | 55.71 | 49.20 | 41.20 | 65.56 | 42.68 | 25.50 | 64.87 | 67.40 | 43.12 | 67.31 | 66.59 | 4.74 | 3.00 |
| MQ | 69.94 | 60.14 | 47.54 | 38.86 | 68.43 | 43.56 | 22.58 | 68.27 | 70.51 | 60.40 | 47.74 | 39.04 | 69.55 | 44.00 | 22.58 | 69.39 | 71.15 | 40.18 | 71.25 | 71.08 | 3.40 | 4.12 |
| **Avg.** | 68.78 | 61.24 | 54.07 | 44.47 | 68.77 | 51.65 | 37.43 | 68.54 | 68.39 | 59.25 | 57.23 | 47.08 | 68.96 | 52.25 | 44.10 | 69.49 | 70.15 | 46.14 | 71.48 | 71.23 | 4.48 | 4.46 |

Table 6.5: BLEU (B) and ROUGE (R) scores for No-NLP, NLP-1, and NLP-3 are included side-by-side for each project. The NLP model improves the scores across the board. For example, for MQ, the No-NLP B-1 and R-1 scores are 69.94 and 68.43, respectively, and the values increase to 70.51 and 69.55 for the NLP-1 model, and furthermore, rise to 71.15 and 71.25 for the NLP-3 model.



| Projects | He *et al.* [145] % (W) | | | | No-NLP % (X) | | | | NLP-1 % (Y) | | | | NLP-3 % (Z) | | | | Improvement % | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | B-1 | B-4 | R-1 | R-L | B-1 | B-4 | R-1 | R-L | B-1 | B-4 | R-1 | R-L | B-1 | B-4 | R-1 | R-L | $X_{(over)}W$ | $Z_{(over)}X$ |
| HD | 36.59 | 16.96 | 36.88 | 36.24 | 51.74 | 30.21 | 56.97 | 56.49 | 52.95 | 31.27 | 58.72 | 58.22 | 53.77 | 32.89 | 59.40 | 58.92 | 54.27 | 4.90 |
| CS | 47.60 | 27.57 | 47.11 | 46.05 | 50.20 | 33.11 | 55.81 | 54.78 | 50.99 | 33.95 | 57.60 | 56.68 | 53.89 | 37.21 | 61.07 | 59.98 | 15.19 | 9.41 |
| HB | 37.69 | 18.28 | 38.47 | 37.71 | 49.70 | 30.06 | 56.98 | 56.59 | 50.31 | 30.75 | 58.50 | 58.08 | 52.32 | 33.67 | 60.00 | 59.61 | 46.30 | 6.35 |
| HI | 40.78 | 23.04 | 40.58 | 40.08 | 51.99 | 36.13 | 54.95 | 53.36 | 52.47 | 36.31 | 55.43 | 53.84 | 54.16 | 38.74 | 58.70 | 56.85 | 34.31 | 6.12 |
| CL | 51.98 | 30.74 | 50.23 | 49.62 | 60.03 | 34.77 | 63.86 | 63.02 | 60.56 | 35.01 | 63.97 | 63.13 | 62.54 | 38.44 | 66.79 | 65.91 | 21.42 | 5.41 |
| **Average** | 43.06 | 23.18 | 42.85 | 42.11 | 52.73 | 32.86 | 57.71 | 56.84 | 53.46 | 33.46 | 58.84 | 57.99 | 55.34 | 36.19 | 61.19 | 60.25 | 32.38 | 6.41 |

Table 6.6: BLEU (B) and ROUGE (R) scores comparison for [145], the LSD from code clone with no modification, *i.e.*, No-NLP, considering only one prior token in prediction, NLP-1, and considering a sequence of three prior tokens, NLP-3. The 'Improvement' column shows the percentage that No-NLP improves on prior work, and how much NLP-3 improves over No-NLP. On average, NLP-3 makes 40.86% improvement over [145] ($Z_{(over)}W$).

preceding the logging statement *'log'*. As such, to employ our approach, we perform initial preprocessing to wrap the *'code'* segment inside a dummy method, and then we utilize LACCP and NLP CC'd for LSD suggestions, and then compare them with the provided *'log'*. Table 6.6 summarizes the scores for [145] and our approach. He *et al.* [145] achieve 36.59 and 37.69 BLEU-1 scores for Hadoop and HBase, respectively. In comparison, our No-NLP approach achieves 51.74 and 49.70 for BLEU-1 scores for Hadoop and HBase, respectively. Similarly, our ROUGE scores outperform prior work. We contribute this higher performance to the more sophisticated search of clone pairs compared to employing the ten preceding lines of the code utilized in [145]. Table 6.6 also provides the NLP-1 and NLP-3 scores, which further improve the No-NLP ones. The NLP model is successful in remembering the general LSD patterns in each project and further enhances the LSD suggestions. Another observation we made is that NLP CC'd values are generally lower on the sample data [33] in Table 6.6 than the values in Table 6.5, as we hypothesize method-level clone detection provides a better context for LSD prediction than selecting the ten preceding lines of code in the sample data [145].

**Finding 3.** *The additional context provided through finding similar code snippets can be borrowed as a starting point for LSD automation and further augmented with deep-learning NLP approaches.*

**Finding 4.** *Our LSD prediction approach, on average, achieves 32.38% improvement over the prior work ($\frac{X}{W}$) and 6.41% improvement over the No-NLP version ($\frac{Z}{X}$).*

## 6.6 Case Study

Yuan *et al.* [325] showed that developers spend a significant amount of time revising logging statements for system dependability tasks, such as postmortem failure analysis. In this case study for the Hadoop project (Listing 6.2), we investigate the code snippets that have logging statements updates or revisions after a problem was detected in the software systems. Then, we try to find code clones based on that snippet of the source code and see if we could have predicted the logging statement description prior to the failure that could have saved engineering time and trial-and-error cycle. We use *'git blame'* to assign commit number and data and time of the commit, and we look for clones in the portion of the code which was developed and checked in prior to the log statement fixes (*i.e.*, for clone detection we rely on the code which was previously developed while the new code is

being composed). We found a JIRA ticket for the YARN subsystem of Hadoop, YARN-985 [136]. Code Snippets 1 and 2 are clone pairs in Listing 6.2. Snippet 2 went through two logging updates in two different *git commits* in the Year 2014, highlighted in orange and red, respectively. These modifications could have been avoided if the engineer developing the logging statement to Code Snippet 2 had access to observe the logging description from its clone pair, Code Snippet 1 and NLP CC'd predictions for LSD suggestions.

*Listing 6.2: Case study from JIRA; two log changes.*

```
1  // Snippet 1, FairScheduler.java.
2  2013-12-13    private synchronized void removeApplicationAttempt
   (
3  2012-07-13        ApplicationAttemptId applicationAttemptId,
4  2014-01-10        RMAppAttemptState rmAppAttemptFinalState,
   boolean keepContainers) {
5  2012-07-13          LOG.info("Application " +
   applicationAttemptId + " is done." +
6  2012-07-13          " finalState=" + rmAppAttemptFinalState);
7  2014-08-12            SchedulerApplication<FSAppAttempt>
   application =
8  2014-01-10            applications.get(applicationAttemptId.
   getApplicationId());
9  2014-08-12            FSAppAttempt attempt = getSchedulerApp(
   applicationAttemptId);
10                       ...
11 // Snippet 2, CapacityScheduler.java.
12 2013-12-13    private synchronized void doneApplicationAttempt(
13 2011-08-18        ApplicationAttemptId applicationAttemptId,
14 2014-01-10        RMAppAttemptState rmAppAttemptFinalState,
   boolean keepContainers) {
15 2014-01-02     LOG.info("Application Attempt " +
   applicationAttemptId + " is done." +
16 2014-09-12      " finalState=" + rmAppAttemptFinalState);
17 2014-01-10       FiCaSchedulerApp attempt =
   getApplicationAttempt(applicationAttemptId);
18 2014-05-22       SchedulerApplication<FiCaSchedulerApp>
   application =
19 2014-01-10       applications.get(applicationAttemptId.
   getApplicationId());
20                       ...
```

## 6.7 Discussion

### 6.7.1 Log Verbosity Level (LVL) and Variables (VAR)

Our approach is reasonably extendable to predict LVL and VAR alongside the LSD suggestion. Since we have access to the source code of the method that we are predicting the logging statement for and its clone pair, a reasonable starting point is to suggest the same LVL as of its clone pair, and then augment it with additional learning approaches such as [193, 54, 203] for more sophisticated LVL prediction. For example, our analysis for the evaluated projects in Figure 6.5 shows that code clones match in their verbosity levels in the range of (92, 97)%. In addition, we also hypothesize that some of the LVL mismatches in the clone pair LVLs are due to the log-related issues [142] that our clone-based approach has uncovered. For VAR prediction, our approach can be augmented with deep learning [209] and static analysis of the code snippet under consideration [326] to include log variables suggestions alongside the predicted LSD.

### 6.7.2 Practicality in Software Engineering

We note that the ideas similar to our approach for automated log generation have been already applied and proven to be effective in adjacent software engineering tasks such as automated *commit message* [302] and *comment* [305] generation. For example, Wei *et al.* [305] used comments of similar code snippets as *'exemplars'* to assist in generating comments for new code snippets. Both papers' ideas and application scenarios are analogous to a large extent to those of our work. Similarly, both approaches utilize BLEU [305, 305] and ROUGE-L [305] scores for evaluating the quality of the auto-generated text.

## 6.8 Threats to validity

We categorize external and internal threats to the validity of our research.

### 6.8.1 External Threats

External threats to the validity reflect on the generalization of our work to other such software projects and programming languages. In this research, we conducted our log

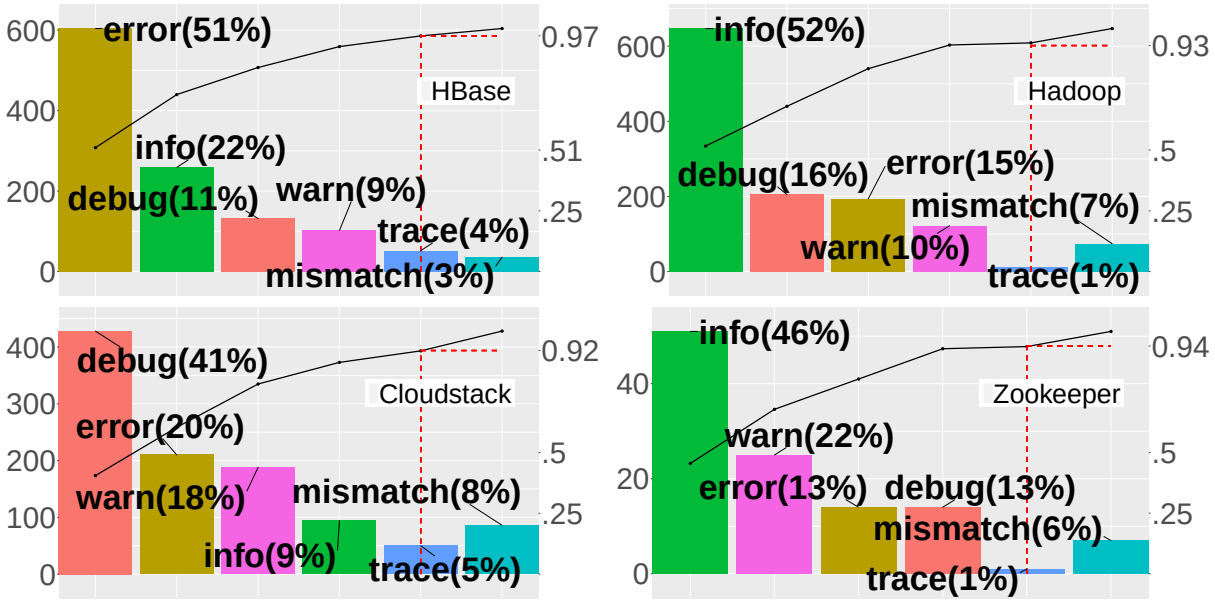



Figure 6.5: Percentage values for each verbosity level. For each project, only a small percentage of clones have a *'mismatch'* in their log verbosity levels.

statement analysis on seven open-source Java projects that are well-established and continuously maintained, and used in prior logging research [78, 145]. We assumed our approach is independent of the underlying programming language that the source code is implemented with. However, since other software systems, and other programming languages, may follow different logging practices, our findings may not accurately extend and generalize to other such systems.

### 6.8.2 Internal Threats

Regarding internal threats, our approach relies on the clone detector to find a clone pair for providing logging suggestions, which implies that we cannot suggest a logging statement for a newly-developed code snippet if it is not similar to a priorly-developed code snippet. In fact, we argue this threat is not exclusive to our approach but also exists for all other log prediction approaches that rely on learning from logged snippets of source code [346, 200], as a learning model can only predict a logging statement for a new code snippet if it can find a feature mapping to its learned logged code base. To mitigate this concern, we suggest curating a database of available open-source code that can be readily parsed and become

available for clone detection. This database could be used to improve the *hit-rate* of the clone detection approach when searching for similar code snippets [155, 145]. Additionally, the architecture of the LSTM model and tuning of its hyperparameters [132] can have an impact on the BLEU and ROUGE scores for different software projects.

## 6.9 Related Work

We categorize the prior work into three main areas: log prediction, code clone detection, and NLP research in software systems.

### 6.9.1 Log Prediction

Yuan *et al.* proposed *ErrorLog* [324], a tool to report error handling code, *i.e.*, *error logging*, such as *catch clauses*, which are not logged, and to improve the code quality and help with failure diagnosis by adding log statements to these unlogged code snippets. Zhao *et al.* [339] introduced *Log20*, a performance-aware tool to inject new logging statements to the source code to disambiguate execution paths. *Log20* introduces a logging mechanism that does not consider developers' logging habits or concerns. Zhu *et al.* [346] proposed *LogAdvisor*, a learning-based framework, for automated logging prediction which aims to learn the frequently occurring logging practices automatically. Their method learns logging practices from existing code repositories for *exception* and *function return-value check* blocks by looking for textual and structural features within these code blocks with logging statements. Jia *et al.* [166] proposed an intention-aware log automation tool called *SmartLog*, which uses an *Intention Description Model* to explore the intention of existing logs. Li *et al.* [200] categorized six block-level logging locations, and Cândido *et al.* [72] performed an exploratory study of log placement with transfer learning for an enterprise software. We discussed how our approach differs from these works during the discussion in Section 6.5.2. Also, our research is the only one that tackles both log location and description automation. Gholamian and Ward [123] showed that code clones follow similar logging patterns and investigated the feasibility of predicting the *"location"* of log statements in an experimental study, but failed to fully observe the clone-detection shortcomings for log prediction. Another research also proposed steps involved in leveraging similar code snippets for log statement prediction [122]. The author also discusses the practicality of their approach during the software's development cycle. Our goal in this study is to improve on the performance of log-aware clone detection and also predict the *"description"* of log statements by utilizing code clones and deep-learning NLP approaches.

### 6.9.2 Code Clone Detection

Source code clone detection is a well-established area of study for software systems, and a significant number of detection techniques and tools have been presented in the literature [261, 47]. Code-clone detection is the task of identifying syntactically exact or similar snippets of source code (with equal semantics) within or between software systems [270, 271] based on contextual features of the source code snippets. We demonstrated that searching for similar code snippets, *i.e.*, clone pairs, is beneficial in automated log statement generation. We initially observed the shortcomings of the generic state-of-the-art clone detection methods [270] for log automation, and then improved on by our log-aware method (LACCP), and proposed an approach to suggest log statements' *description* (NLP CC'd).

### 6.9.3 NLP in Software Systems

Prior research has widely utilized natural language attributes for various applications in software engineering. For example, natural language exists in software source code and identifier names, design documents, bug reports [55], and code suggestion [66]. To enable NLP for software systems, prior research [119, 155] has shown the source code is redundant and repetitive, which can be utilized to model the source code with n-gram language models. Tu *et al.* [298] further explored the localness of software characteristics in order to utilize regularities that can be captured in a locally estimated cache and leveraged for software engineering tasks. Most recently, research has shown that logging statements' descriptions [145] and execution log files [126] manifest natural language features, similar to other software artifacts, such as source code itself. Inspired by the prior NLP research, in this work, we utilize a deep-learning NLP model for logging statements' description prediction, and outperform the results in prior work [145].

## 6.10 Conclusions and Future Directions

Software developers insert logging statements in the source code in various places to improve the software development process and its diagnosability. Nevertheless, this process is currently manual, and it does lack a unified guideline for the location and content of log statements. In this chapter, with the goal of log automation, we presented a study on the *location* and *description* of logging statements in open-source Java projects by applying similar code snippets and NLP models. We initially improved the performance

of the log-aware code clone detector (LACCP) by 15.6% compared to Oreo, and then augmented the performance of log description prediction with the deep learning natural language processing approaches. We experimented on seven open-source Java systems, and our analysis shows that by utilizing log-aware clone detection and NLP, our hybrid model, (*NLP CC'd*), achieves 40.86% higher performance on BLEU and ROUGE scores for predicting LSDs when compared to the prior research ($Z_{(over)}W$), and achieves 6.41% improvement over the No-NLP version ($Z_{(over)}X$). We also included a case study of logging issues in Hadoop, a discussion on the applicability of our approach and prediction of the log verbosity level and its variables, and threats to the validity of our research. As future work, we look into further incorporating the source code surrounding the logging statements for additional context in log automation.

## 6.11 Repository Explained

We provide a repository [12] to make our data available. The main folders are *LACCPlus* and *NLPCCd* for RQ1 and RQ2, respectively. Under each folder, there are subfolders for each software project, *e.g.*, *Zookeeper*. Inside each subfolder, we have provided clone pairs for methods that we have examined in our study. The naming convention for each method consists of its id (*i.e.*, method id) and an index for each method snippet, such that *(id_1, id_2)* forms a clone pair.

# Part IV
# Log Natural Language Processing

# Chapter 7

# What Distributed Systems Say: A Study of Seven Spark Application Logs

**Abstract-** Execution logs are a crucial medium as they record runtime information of software systems. Although extensive logs are helpful to provide valuable details to identify the root cause in postmortem analysis in case of a failure, this may also incur performance overhead and storage cost. Therefore, in this research, we present the result of our experimental study on seven Spark benchmarks to illustrate the impact of different logging verbosity levels on the execution time and storage cost of distributed software systems. We also evaluate the log effectiveness and the information gain values, and study the changes in performance and the generated logs for each benchmark with various types of distributed system failures. Our research draws insightful findings for developers and practitioners on how to set up and utilize their distributed systems to benefit from the execution logs.

**Keywords:**
logging statement, log verbosity level, log4j, logging cost analysis, information gain, entropy, distributed systems, system failure, Apache Spark, HDFS

**An earlier version of this chapter is published in the International Symposium on Reliable Distributed Systems (SRDS) 2021 [127].**

## 7.1 Introduction and Motivation

The rapid growth of processing requirements and data scale in computing systems has contributed to the development and adaptation of large-scale, parallel, and distributed computation and storage platforms, *e.g.*, Apache Spark and Hadoop Distributed File System (HDFS). Laterally, as the size of the data and computing systems grow, and they become more distributed in nature, evaluating their reliability and performance becomes more daunting. As such, execution log files and instrumentation of the source code are important origins of information for dependability analysis and gaining insight into the runtime state of the system. Execution logs have advantages over instrumentation, as they are readily available, do not require access to the source code, and do not introduce perturbation [218]. However, instrumentation requires access to the source code, and it incurs perturbation due to the added instrumentation code.

Logging is an important integral part of the software development process to record necessary run-time information [117, 141]. Software developers insert logging statements into the source code to record a wealth of information such as variable values, state of the system, and error messages. Developers and system operators use this information for different purposes, among them failure and performance diagnosis [311, 104]. Although logging has proven benefits, it can incur system costs. Excessive logging can cause system overhead, such as CPU and I/O consumption. Contrarily, logging too little may miss important information and degrade the usefulness of execution logs [117]. Authors of [149] described a typical online system at Microsoft that could produce execution logs in the terabyte order-of-magnitude per day. As such, this high volume of logs can impair the quality of service for such systems. To address the trade-offs associated with the overhead of logging, well-known libraries, such as Apache Log4j [15] and SLF4j [26], provide facilities for different levels and granularities of logging. The libraries provide different verbosity levels to dynamically control the number of logging statements being ultimately outputted to the log file on the storage medium. As each logging statement comes with a verbosity level, the logging library filters log messages by comparing the log statement's level with the dynamic log level specified by the user. Log4j has six verbosity levels available to the developers by default: *fatal, error, warn, info, debug, and trace*. Figure 7.1 shows an example of a logging statement from Spark with *info* verbosity level and its end product in the log file. In addition, each logging statement consists of a constant part, *i.e.*, *"Executor added: on with core(s)"*, and a variable part, *i.e.*, *"fullId"*.

Log levels represent a measure of the importance of the messages. For example, less verbose levels (*i.e.*, *fatal*, *error*, and *warn*) are used to warn the user when a potential problem happens in the system. On the other side, more verbose levels such as *info*, *debug*,

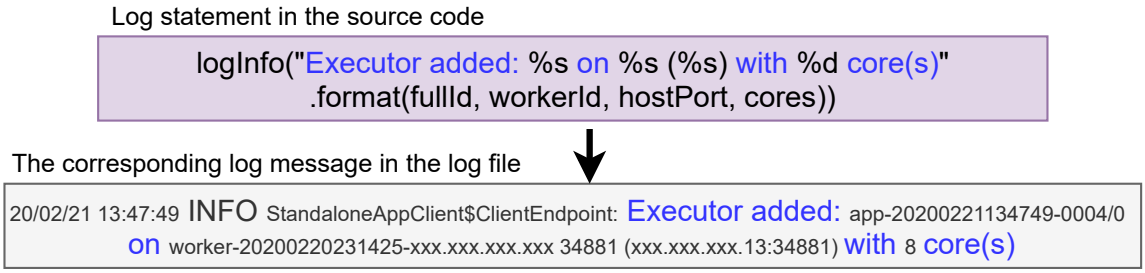



Figure 7.1: Log statement and end product in the log file.

and *trace* are utilized to track more general system events and information or detailed debugging. Considering the flexibility that each log level brings, our goal in this research is to quantitatively measure the cost, in terms of storage, execution overhead, and information gain (IG) of log files while the distributed system is running under different log verbosity levels. Ultimately, we aim to reach a guideline on implications for developers and practitioners on how to utilize the logs in different verbosity level decisions while developing or operating distributed software systems in normal scenarios and in presence of failures. We guide our research with the following research questions (RQs):

- **RQ1:** *what is the quantitative cost of logging in terms of computation time (CT) and storage overhead (SO)?* (§7.3)
- **RQ2:** *how much information is gained from different log verbosity levels (VLs)?* (§7.4)
- **RQ3:** *how the characteristics of logs change with distributed failures, i.e., distributed computation and storage failure? Does the entropy of logs increase when a failure happens?* (§7.5)

For each RQ, we discuss the practical findings of our analysis and their implications for developers and practitioners on how to utilize the execution logs. Our research provides insight on how to choose the level of logging, and ultimately control the amount of generated logs and the information gain, and how the failures can be detected with entropy values. In addition, we provide a discussion on our findings and the implications for future improvements in distributed systems and their scheduling in case of system failures (§7.5.4). With the motivation of helping developers and practitioners to gain more insight into the content of execution logs, and to make more deliberate logging level decisions, we pursue the following contributions in this chapter: **(1)** We evaluate the performance and cost of logging for Spark under a set of batch and iterative workloads with different

characteristics to calculate the overall execution time overhead and volume of generated logs (**RQ1**). **(2)** We calculate the information gained from the log files in different VLs based on their entropies and natural language processing (NLP) of logs with n-gram models and provide insights on how to make logging level decisions based on the observed cost and information gain from the log files (**RQ2**). **(3)** We introduce a comprehensive set of distributed system failures and evaluate the changes in execution log characteristics and entropy values, and provide insights on practical outcomes of our analysis for how to utilize execution logs to pinpoint failures (**RQ3**). Lastly, we release our labeled failure logs to encourage and enable further research in this field [6].

## 7.2 Approach and Setup

In this section, we present our approach and characterize the systems, their configurations, and the workloads that we use to conduct our study. Figure 7.2 outlines the steps involved in our study. We categorize the logging cost into two system aspects: 1) execution overhead and 2) storage cost. We run seven Spark benchmarks with different log verbosity levels and calculate the execution times and the size of the generated logs. We then utilize Shannon's entropy theory [281] and n-gram models [9] to measure the information gain by calculating entropies for different log levels with and without failures.

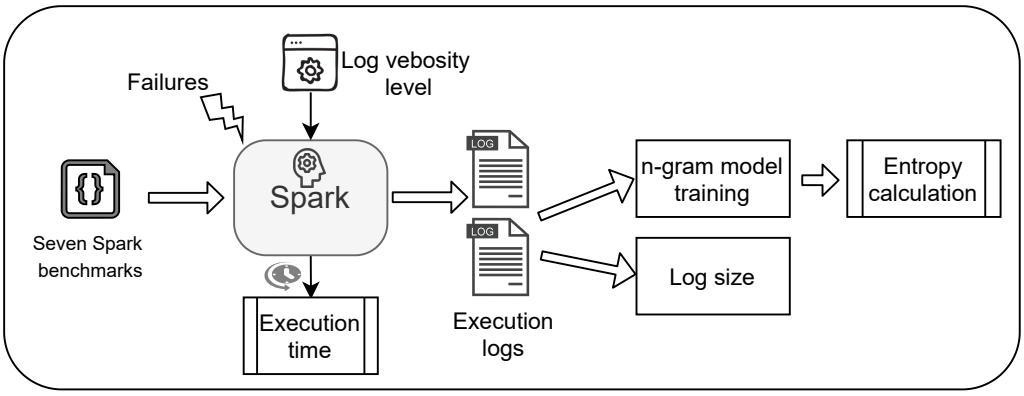



Figure 7.2: Our approach for measuring the cost and effectiveness of the logs.

**Apache Spark.** Since its introduction, Spark has been widely adopted as a big-data, distributed, and parallel processing framework. Spark builds upon Hadoop's MapReduce model and brings extra flexibility and improved performance. Additionally, Spark provides interfaces to other big-data platforms such as Hadoop's distributed file system, HDFS. To

achieve higher performance and as an improvement to Hadoop MapReduce, Spark utilizes Resilient Distributed Datasets (RDDs) [327], which retain the intermediate results in main memory, and therefore, reduces the overhead caused by the disk and network [328]. This optimization benefits Spark the most in iterative tasks (*e.g.*, Transitive Closure), as the following stages of the task rely on the intermediate results from the prior stages. Because of its improved performance and widespread use, we deploy a Spark cluster to perform our study.

**Hardware.** Table 7.1 and Fig. 7.3 show the main hardware characteristics and the architecture of our deployed cluster, respectively. Each node in the cluster has 12 (12*2 hyper-threaded) cores, 32 GBs of memory, and 2 storage disks of 1 TB each. We evaluate the benchmarks on a cluster of 4 commodity machines illustrated in Fig 7.3. Each machine is equipped with dual 2.40GHz Intel Xeon E5-2620 CPUs, supporting a total of 24 hyperthreads per machine and a 1Gbps NIC. All servers run Ubuntu Server 16.04.6 LTS 64-bit with kernel version 4.4.0-159-generic.

| Name | Role | Cores | Memory | Disk | Local IP |
|---|---|---|---|---|---|
| styx01 | Master/NameNode | 12 (24 HT) | 64 GB | 2*1 TB | 192.168.210.11 |
| styx02 | Worker/DataNode | 12 (24 HT) | 64 GB | 2*1 TB | 192.168.210.12 |
| styx03 | Worker/DataNode | 12 (24 HT) | 64 GB | 2*1 TB | 192.168.210.13 |
| styx04 | Worker/DataNode | 12 (24 HT) | 64 GB | 2*1 TB | 192.168.210.14 |

Table 7.1: Styx cluster for Spark computation and HDFS.

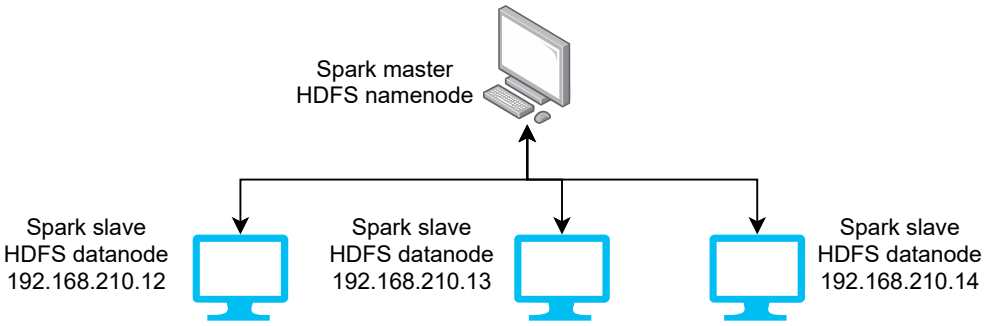



Figure 7.3: Design of the distributed cluster, consisting of one master/name node and three slave/data nodes.

**Framework setup.** In our experiments, we use Spark 2.4.4 and Hadoop 2.9.2. We use *styx01* as a dedicated server for the Spark *master* and HDFS *name node*, while having one Spark *slave* and HDFS *data node* on each of the three other machines, *styx02, styx03,*

| Framework | Parameter | Value |
|---|---|---|
| HDFS | block size | 128 MBs |
| HDFS | Replication factor | 3 |
| Spark | Worker/Executor per node | 1 |
| Spark | Cores per executor | 24 HT |
| Spark | Memory per executor | 32 GBs |
| Spark | Driver memory | 10 GBs |

Table 7.2: Main Spark and HDFS settings.

*and styx04*. Hadoop file system block size is 128 MBs and replication is set to 3. We set up each Spark slave to use 12 available cores and up to half of the available memory *(i.e.*, 32 GBs out of 64 GBs) on each of the nodes. The frameworks have been carefully configured according to their corresponding user guides and the characteristics of the system (*e.g.*, number of CPU cores and memory size). Table 7.2 summarizes the related configurations for HDFS and Spark nodes.

**Benchmarks.** In this research, we experiment on seven different Spark benchmarks and provide a brief explanation of each one in the following:

❶ **WordCount (WC)**, which counts the number of times each word appears in the input dataset. By applying transformations such as *'.reduceByKey()'* on RDDs, WordCount outputs a dataset of (word, value) pairs, saved to a file on HDFS. WordCount is a popular Spark's benchmark that allows us to assess CPU and I/O costs associated with different levels of logging.

❷ **TeraSort (TS),** which sorts randomly generated rows of key-value (KV) pairs with each KV being 100 bytes. The TeraSort implementation and its random data generator engine, TeraGen, are both adopted from [41].

❸ **TransitiveClosure (TC),** checks and implements linear transitive closure (LTC) on a graph, iteratively. For example, if *x, y, and z* are three vertices, and *(x,y)* and *(y,z)* represent edges between *x and y*, and *y and z*, respectively, for satisfying the transitive closure property, a new edge is added between *x and z*. LTC grows paths by one edge, by joining the graph's edges with the already-discovered paths in each iteration. TC is an iterative CPU-intensive workload.

❹ **PageRank (PR)** is an iterative graph algorithm that ranks URLs by considering the number and rank of URLs referring to it. For example, the more URLs with higher ranks refer to a URL under consideration (URLUC), the URLUC's rank increases.

For the PageRank's implementation, we use the implementation provided with the Spark's example package.

❺ **TestDFSIO (DF),** which is a benchmark designed to evaluate the I/O (read/write) performance by using Spark's tasks to read and write multiple files in parallel. The benchmark aims to read and write an even amount of data to HDFS on each node in the cluster. The implementation is adapted from [42].

❻ **GradientBoostingClassificationTrees (GC),** which is a machine learning algorithm for classification, that generates a prediction model as an ensemble of decision trees. In this use case, the number of classes is set to two, the depth of the trees is set to five, and we perform 200 iterations for the model training.

❼ **LinearDiscriminantAnalysisClustering (LD),** which implements LDA clustering algorithm, *i.e.*, unlabeled data, that clusters the input documents into three different topics.

Table 7.3 summarizes the benchmarks used in the experiments, along with their characterization such as CPU or I/O (disk and network) intensive, and if the computation happens iteratively. The sizes of the input datasets are also shown, and we refer to the benchmarks with their abbreviation in the rest of the chapter, as shown in Table 7.3. During the benchmark selection, we were deliberate to include a variety of workloads such as Spark's conventional benchmarks, *e.g.*, WC and PR, and machine learning ones, *e.g.*, GC and LD.

| Benchmark (abbrv.) | Task type | Input data size | Notes |
|---|---|---|---|
| WordCount (WC) | CPU and I/O intensive | 52 GBs | The output is pairs of (word, count) written to HDFS. |
| TeraSort (TS) | Iterative, I/O, and CPU intensive | 2 GBs | Sorts randomly generated (key, value) pairs, and the size of each pair is 100 bytes. |
| TransitiveClosure (TC) | Iterative and CPU intensive | small (few KBs) | Calculates the transitive closure on a randomly generated graph with 200 edges and 100 vertices. |
| PageRank (PR) | Iterative and CPU intensive | 40 MBs | Ranks web pages based on their popularity. |
| DFSIO (DF) | I/O intensive | 20 × 1 GB | Writes and then reads 20 files of one GB each to HDFS. |
| GradientBoosting Classification Trees (GC) | Iterative and CPU intensive | small (205 KBs) | Trains 200 decision trees with the depth of five for classification of a decision problem, *i.e.*, yes (1) or no (0). |
| LinearDiscriminant Analysis Clustering (LD) | Iterative and CPU intensive | 21 MBs | Clusters the input data into three topics using LDA. |

Table 7.3: Benchmark characteristics.

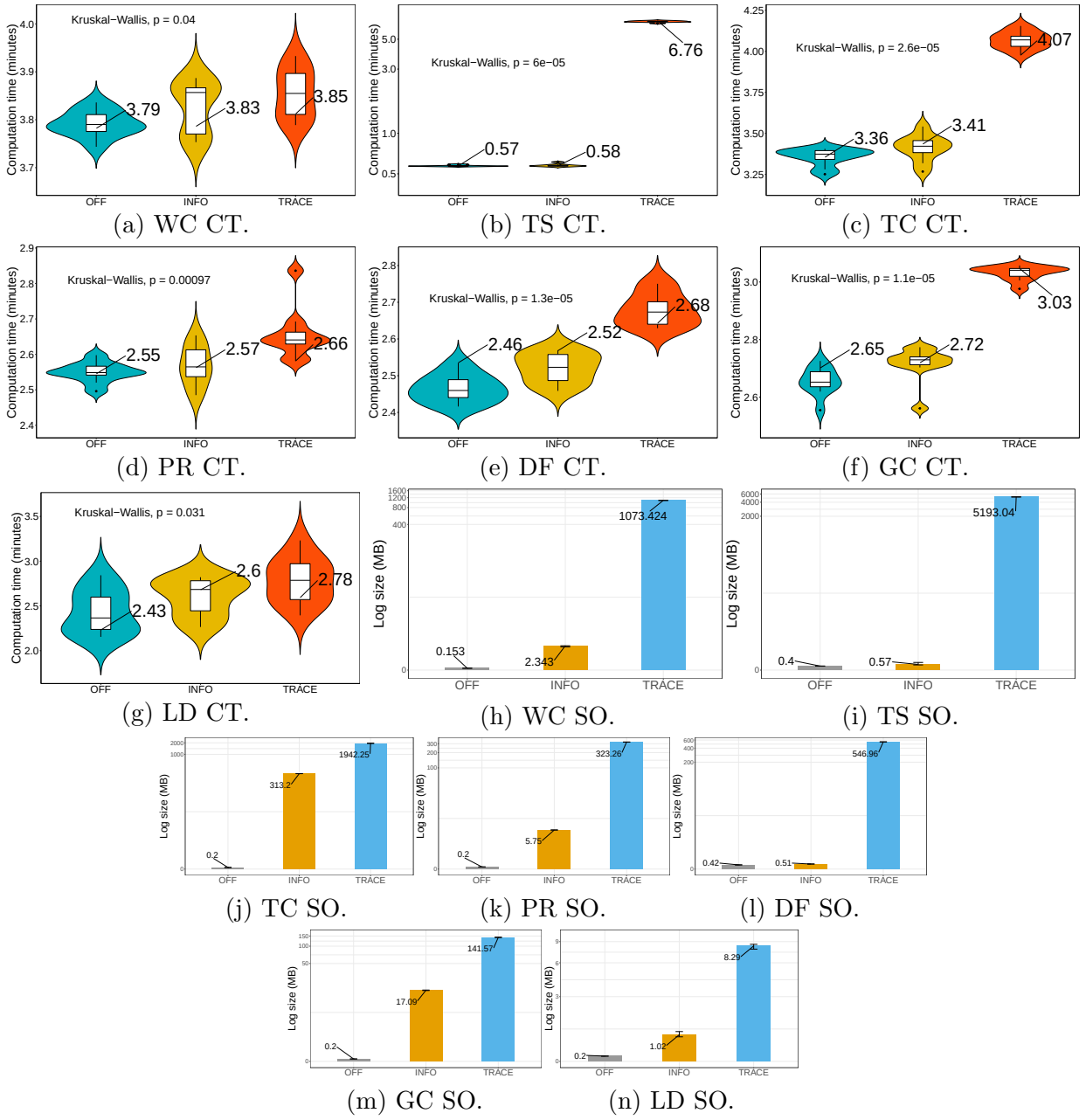



Figure 7.4: Computation time (CT) and storage overhead (SO) for *WordCount* and *K-Means* tasks.

## 7.3 RQ1: Cost of Logging

There exists various qualitative and quantitative metrics for assessing logging cost. Quantitative metrics consider the overhead of logging on different subsystems of the computing systems, *e.g.*, CPU and I/O overhead [104], and qualitative metrics assess the cost of logging in terms of developer and user experience, such as the cost of revealing private information through logs [192]. In our work, we focus on quantitative measurement of the logging cost and for this purpose, we conduct a set of experiments to evaluate the impact of logging verbosity level on the size of the generated logs, as well as the effect on the performance of the Spark. To measure the cost of logging, we run multiple Spark benchmarks on our commodity cluster and calculate the logging cost in terms of the size of the generated log and the execution time for each benchmark. The measured computation time and storage values in this section serve as a baseline for comparison on further RQs.

### 7.3.1 Computation time (CT)

Figures 7.4a-7.4g show the violin chart and interquartile range with mean values (noted in text) for execution time of different benchmarks. We also perform Kruskal-Wallis test [182] to reject the null hypothesis and ensure statistically significant values, *i.e.*, $p \leq 0.05$. Violin chart improves on boxplot chart by providing the width in the graph as the density of points in the experiments. The vertical axes represent the execution time in minutes, and the horizontal axes show different verbosity levels (VLs). We select to show *info* and *trace* VLs as the former is generally enabled by default and presents the normal mode of logging, and the latter represents the maximum amount of logging which is widely utilized during failure diagnosis [83, 345]. This provides a picture of the lower and upper bounds of logging. Besides, we present the data for logging *off* to present the other end of the spectrum (least amount of logging) compared to the *trace* level (most amount of logging). The overall trend shows the more verbose VLs result in higher execution times. Because *trace* VL enables more detailed logging and executes additional lines of code (and more I/O system calls), it incurs a minor but noticeable execution time overhead across different benchmarks, when compared to *info* level.

### 7.3.2 Storage overhead (SO)

Figures 7.4h-7.4n show the size of generated execution logs for different benchmarks. The vertical axes represent the size of the log file in MB, and the horizontal axes show different

verbosity levels. The sizes show the aggregated logs for all the nodes (*i.e.,* master and workers) of the Spark cluster. Similar to computation times, *trace* level enables more detailed logging, which results in significantly higher volumes of log data when compared with less verbose log levels.

**Results review.** Based on the conducted experiments, we observe that the execution time of the benchmarks increases as the log level becomes more verbose. This observation is congruent with the knowledge that the execution of logging messages infers extra CPU, I/O, and storage cost. Overall, excluding TS, we see on average 8.01% overhead in execution time when the *trace* log level is enabled versus the *info* level. For the log file size, we notice on average a ∼268X increase in the volume of the generated logs for the *trace* log level versus the *info* verbosity level. We did a further investigation to better understand the ∼12X increase in computation time for TS, and we observed the significant amount of generated logs for TS in *trace* level when compared to *info* (*i.e.*, 5 GBs *vs.* 0.5 MBs). Because TS is a CPU-intensive benchmark, we rationalize that its CT suffers noticeably due to the significant amount of logs outputted in the *trace* level. Comparing CT and SO values for different benchmarks, we observe that the amount of generated logs in different VLs is benchmark dependent, and CPU-intensive applications (*e.g.*, TS, TC) observe a higher slowdown due to more verbose logging.

**SO mitigation.** Prior work has shown that due to the high level of repetitiveness in log files, they can benefit from large compression ratios, up to 84% [319]. Therefore, the noticeable difference in storage cost can be mitigated by the compression of log files to ∼43X.

**Finding 1.** *Overall, we observe on average 8.01% and ∼268X overhead in the execution time and storage when the trace log level is enabled versus the info level, respectively, and CPU-intensive workloads suffer more from a higher degree of logging.*
**Implications.** *Considering the trade-offs, if the worst-case 8.01% execution time is acceptable, by utilizing log rolling, compression, and continuous achieving, the storage overhead of more verbose logging can be further lowered.*

### 7.3.3 RAM Disk

We used the hard disk drive (HDD - TOSHIBA MG04ACA200E - 7,200 RPM) as the storage medium for collecting the logs. Because we observed significant degradation for the performance of some of the CPU-intensive applications such as TeraSort when the *trace* level is enabled, we further investigate the impact of utilizing faster storage systems

for log collection. Because developers and practitioners mostly utilize trace log level for debugging, we rationalize that the debugging data can be saved in memory temporarily to expedite the debugging process and the final debugging outcome can be transferred to the disk when the debugging is finalized. In addition, although memory storage is volatile and there is a risk of debugging data loss due to power outage, we presume this risk is manageable as we are concerned with debugging data in contrast to the actual execution logs, and the experiments can be repeated in case the debugging data is lost. Additionally, as new storage technologies become faster, *e.g.,* solid-state drive (SSD), and its latency edges closer to the main memory speed, this data point shows the maximum potential improvement that comes in from a faster storage paradigm, *i.e.*, a latency lower bound and a *'hypothetical'* storage medium that is as fast as the main memory. Table 7.4 compares the CT values for HDD versus *RAM Disk* for benchmarks that we observed a noticeable increase in CT when *trace* level is enabled. RAM Disk is a utility that allows us to map a portion of RAM as disk space and redirect benchmark logs to the space on RAM. Our goal is to show how much of the extra CT introduced because of the slow storage medium can be recovered by leveraging a faster storage medium, assuming non-volatile storage mediums become as fast as RAMs.

| CT (min) | TS | TC | PR | DF | GC | LD |
|---|---|---|---|---|---|---|
| **HDD** | 7.40 | 4.07 | 2.66 | 2.68 | 3.03 | 2.78 |
| **RAM Disk** | 5.73 | 3.99 | 2.61 | 2.64 | 2.99 | 2.68 |
| **CT reduction (%)** | 22.62 | 2.05 | 2.02 | 1.38 | 1.19 | 3.60 |

Table 7.4: Computation time values for RAM Disk *vs.* HDD for *trace* level.

**Finding 2.** *TeraSort, which generates a significantly higher amount of logs in trace level compared to info level, shows the highest CT reduction while using RAM Disk.*
**Implications.** *Faster storage mediums can mitigate some of the overhead associated with logging for CPU-intensive workloads that generate a significant amount of logs in more verbose log levels.*

## 7.4 RQ2: Log Effectiveness

In RQ2, we evaluate the relationship between the log verbosity levels and their effectiveness. Although more verbose logs are used generally for debugging, there has not been any effort

to quantitatively assess the effectiveness of logs in more verbose levels. In other words, although the common perception is that a higher degree of logging translates to more effectiveness of the logs, this assumption might not completely hold true. For this purpose, we introduce a new metric for calculating the effectiveness of logs based on entropy values and investigate whether or not more verbose VLs are more effective.

**Log effectiveness (LE).** LE is a quantitative measure of logs' effectiveness in achieving their goals, which is mainly problem diagnosis and troubleshooting. For example, Yuan *et al.* [325] showed that in their experiments when log statements exist, developers could diagnose system problems *2.2X* faster compared to not having the logs. In this study, LE is directly related to Entropy (*i.e.*, $LE \propto Entropy$) that we clarify in the following. As illustrated in Figure 7.1, log statements consist of two parts: static and dynamic content. Static content of log statements originates from the source code, and dynamic content is the value of variables that are printed in the log files as the system is running. As such, the dynamic content of the logs can be different in each iteration, whereas the static part is unchanged and has the same value in every iteration of the program. Therefore, for more verbose VLs to be more effective than less verbose ones, they should result in higher Entropy values and information gain (IG), as this signifies more unique runtime content. In other words, higher dynamic content translates to more runtime information and value of variables, which is positively related to higher values of entropy, IG, and LE:

$$(\uparrow Dynamic\ content) \rightarrow (\uparrow Entropy) \rightarrow (\uparrow IG) \rightarrow (\uparrow LE)$$

**Shannon's entropy**. We use entropy as a metric to measure the dynamic content and effectiveness of the log records. Shannon's entropy [281] is used to measure the amount of information that is contained in an information source (*e.g.*, a text file). Entropy is calculated as: $H = -\sum_{n=1}^{N} p(i) \log_2 p(i)$, where $p(i)$ is the probability of a possible character happening in the log data [281]. The more random (*i.e.*, less repetitive) the content of the log file, the higher the entropy. For the purpose of experimentation, we focus on the *info* and *trace* log levels as they are used during the deployment and development of the software, respectively, to gain insight from the end user and developer perspectives. Because Spark generates only a few MBs of logs in *info* level for some benchmarks and to perform the experimentation on equal log sizes, we randomly sample 1 MB size of Spark's logs for both *trace* and *info* levels for each benchmark and measure the entropies. Table 7.5 shows the entropy values per character for log files in *trace* and *info* verbosity levels. The character-level entropy values are slightly higher for the *trace* log level, which partially signifies higher information gain (IG) and less repetitiveness for this level.

| Entropy | WC | TS | TC | PR | DF | GC | LD |
|---|---|---|---|---|---|---|---|
| **Info** | 5.24 | 5.31 | 5.16 | 5.21 | 5.26 | 5.23 | 5.26 |
| **Trace** | 5.41 | 5.39 | 5.38 | 5.40 | 5.37 | 5.34 | 5.40 |

Table 7.5: Shannon's entropies for *info* and *trace* for various applications.

**N-gram model.** Although character level entropy explains the randomness of single characters in logs, it does not provide insight on the sentence level repetitiveness of log messages. It is more reasonable to calculate the entropies for a sequence of words, as log statements are inserted as a sequence of tokens (*i.e.*, words and variables) in the source code. To accommodate for a sequence of words, which bears higher semantic meanings for log messages, entropy is also used for a sequence of grams (*i.e.*, words or tokens), such as calculating the probabilities of a sequence of tokens in the English language. For this purpose, prior research has suggested the use of n-gram models [155], to capture the repetitiveness of a sequence of words. The n-gram model captures the probability distribution of the log data, and once trained, it can predict the probability distribution of the next token in new log sequences by utilizing order-n Markov model approximation. This approximation considers the probability of $i_{th}$ element in the sequence of $n$ tokens to be predicted based on $n-1$ preceding tokens [168]. Therefore, we can estimate the probability of $a_i$ succeeding tokens $a_{i-1}, a_{i-2}, ..., a_{i-n+1}$ with:

$$p(a_i|a_{i-1}a_{i-2}...a_{i-n+1}) = \frac{count(a_i a_{i-1}a_{i-2}...a_{i-n+1})}{count(a_{i-1}a_{i-2}...a_{i-n+1})} \tag{7.1}$$

Based on this model, the entropy for a sequence of tokens is:

$$H = -\frac{1}{N}\sum_{n=1}^{N} log\ p(a_i|a_{i-1}a_{i-2}...a_{i-n+1}) \tag{7.2}$$

To measure the sentence-level information gain from the logs, we evaluate the entropy of a sequence of log tokens in both *info* and *trace* levels with n-gram models and compare it with common English text such as *Gutenberg* [10] and *Wiki*. Gutenberg is a collection of English books, and Wiki is the English articles from Wikipedia. To train and test the n-gram models on the sequences of logs, we randomly sample 1 MB of data from each benchmark and perform a 90%-10% train-test split. We run ten-fold cross-validation to avoid overfitting [239] and plot the average entropies of 10 iterations for n-gram models in the range of $n \in (1, 8)$ in Figure 7.5. English text entropies stabilize around eight as the size of n-gram increases, and the median values for *trace* and *info* stabilize at 0.975 and

0.982, respectively. Our experiment reveals that English text has higher entropy than log files, and hence it has less repetitiveness, which is also observed in prior research on the naturalness of software artifacts [155, 298, 120, 126]. Lower baseline entropy of software logs compared to natural language text is beneficial as it results in *'distinguishable'* entropy changes while detecting anomalous log lines (*i.e.*, peaks in entropy values) [126], which can be utilized for log failure detection. Interestingly enough, our comparison shows, in Spark's case, the n-gram entropies are comparable for *trace* level when compared to the *info* level. This suggests that although *trace* level logging results in a higher volume of logs, *trace* log sequences are *not necessarily* less repetitive, and *trace* does not benefit from noticeable higher information gain values, as IG from an event (*e.g.*, log event) is directly related to its entropy [246]. In addition, a higher amount of repetition that results in larger log files might decrease their effectiveness. Redundancy is an undesirable feature of logs since it adds noise to the log files and complicates the understanding of the program's behavior and hinders problem diagnosis through logs [325, 140].

**Finding 3.** *Although trace level generates a larger volume of logs, trace data does not provide a noticeable higher entropy, and hence, does not necessarily carry higher information gain and effectiveness when compared to less verbose log levels.*
**Implications.** *We presume trace logs show comparable IG to info because they contain higher repetition rather than unique dynamic values.*

## 7.5 RQ3: Failure Assessment

Logs are widely utilized in failure detection and performance diagnosis [311, 104]. Therefore, in this section, we study the effectiveness of the information gain approach in system failure detection. To evaluate the effect of system failures on the generated logs, we design a framework to inject different types of distributed failures and measure their impact on logs and how the IG approach can be applied to extract log lines related to the failures. As numerous failure scenarios exist, our goal is not to provide a comprehensive list, but to investigate common failures in a distributed environment. We categorize the distributed failures in four main categories:

1. **Compute node failure** happens when a compute resource becomes unavailable. We synthesize this scenario by terminating one of the Spark's worker nodes.

2. **Storage node failure** in a distributed environment happens when a storage medium becomes unavailable. As we utilize HDFS with the replication factor of three, the

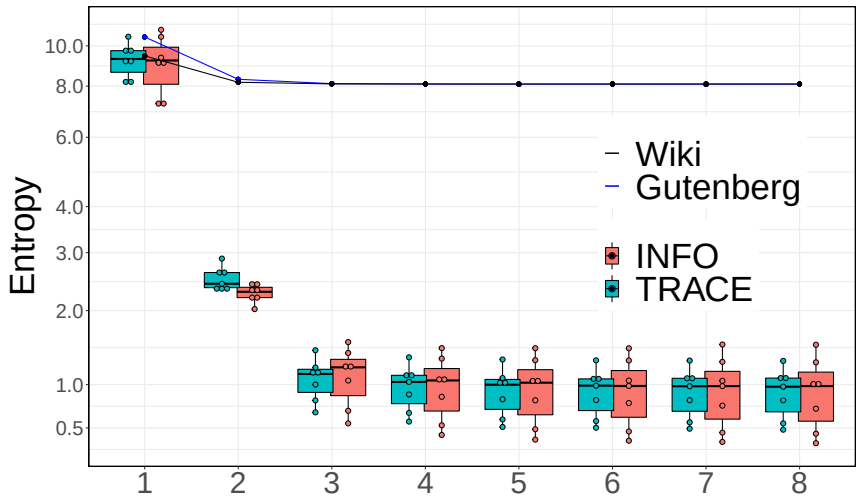



Figure 7.5: Entropy for n-gram models for Spark logs and English text.

integrity of the data remains intact in case of a single node failure, however, the latency of reads and writes to the storage will increase for some compute nodes that require access to data on non-local HDFS nodes. We synthesize this failure by terminating one of the HDFS data nodes.

3. **Communication interference**, which resembles a scenario in the distributed network with variable latency and a probability of packet loss. This category can be initially observed as a performance degradation, and eventually may lead to a complete failure if the communication delay between distributed compute and storage nodes surpasses a system's predefined timeout.

4. **Combined failure** resembles a scenario in which multiple nodes become unavailable simultaneously for various reasons such as power outages. We simulate this scenario by terminating a cluster node that hosts both Spark compute and HDFS storage nodes.

With this failure categorization, our goal is to observe the changes in the content of the logs files and apply information gain approaches to detect failures. The hypothesis is that we should observe a higher information gain during a failure, as the failure related logs should resemble different dynamic content. As such, we evaluate the entropy of logs during their normal and abnormal (*i.e.*, failure) time intervals. Because we noticed comparable entropy values for *info* and *trace* log levels (Figure 7.5), in the following, we focus on *info* verbosity level as it is the default log level during the deployment, and, additionally,

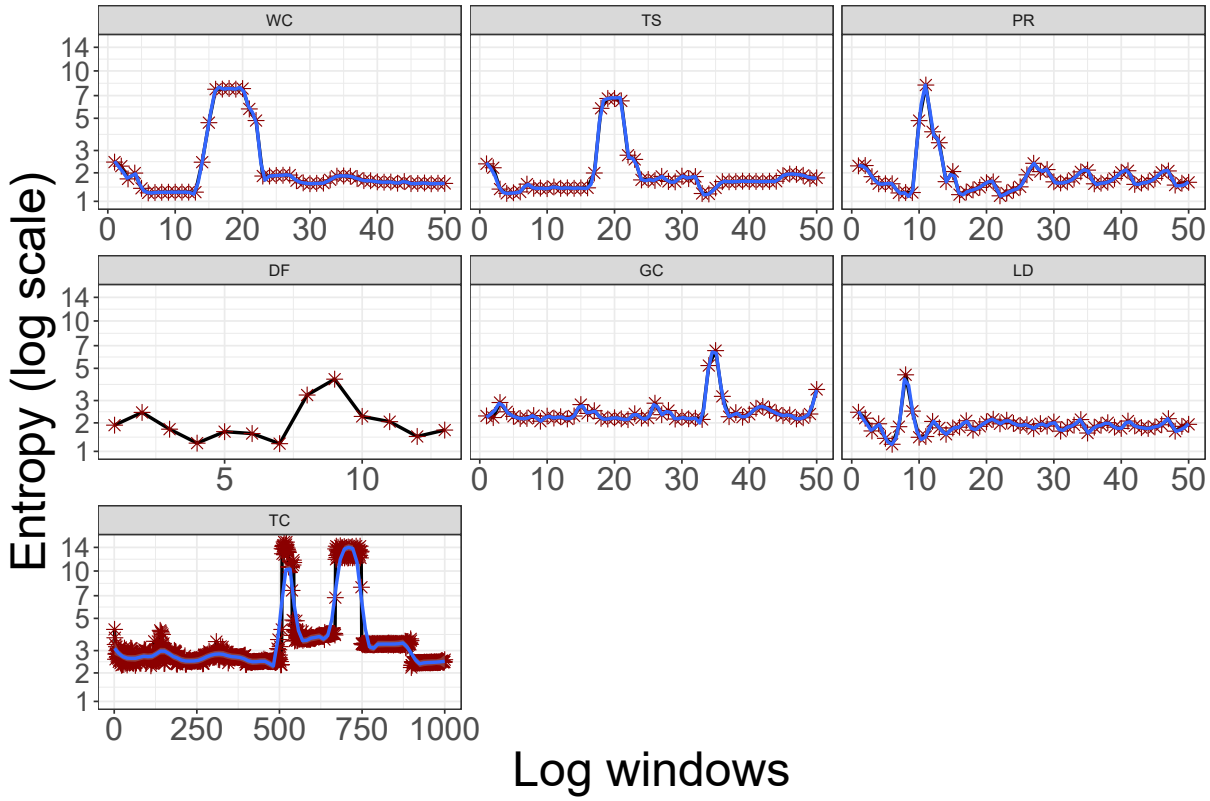



Figure 7.6: Entropy values for log windows for different applications with Spark's compute node failure.

storage overhead of logs becomes more manageable. We run each Spark's benchmark in *info* level for ten iterations with and without the aforementioned failures and evaluate the changes in execution time and the storage overhead for the generated logs. In addition, we also evaluate the changes in information gain (entropy) with the normal and failure logs. For entropy calculation, the n-gram model is trained on the normal execution runs (*i.e.*, without failures) and tested on runs with failures. We choose $n = 5$ for the n-gram model, as according to Figure 7.5, entropy values are stabilized for $n \geq 5$.

### 7.5.1 Compute Node Failure

Figure 7.6 shows the entropy values over time for different benchmarks. The x-axis shows sequential log windows of 4 KBs in size, as suggested by prior work for log analysis time window ([126]), and the y-axis shows the entropies. As the size of the generated logs varies

for each benchmark, we show the timeline of entropy changes for each benchmark from the start to the end of its execution. The spikes in the entropy values are the manifestation of failures in the Spark's logs. Once a failure happens, the system first detects the failure and then plans a set of *recuperating actions* to recover from the failure. For example, for a distributed system such as Spark, the task manager resubmits the tasks previously assigned to a failed node to other nodes in the distributed cluster. This results in several log lines in the log files, which we call *failure manifestation log region*, that have higher than normal entropy values. As the failure happens, the benchmarks also show noticeable prolonged computation time and additional logs compared to the baseline scenario (*i.e.*, with no failure in Figure 7.4).

**Detailed analysis of TransitiveClosure.** We observed that failures can result in different manifestations in the execution logs, and the manifestation can be relatively benchmark dependent. To provide further insight, we review the interesting scenario of a Spark compute node (CN) failure for the TransitiveClosure benchmark, which goes through the following four failure stages: (**S1**) **Failure detection**. Upon a CN failure, the Spark's Master (SM) observes this as *"a CN has been disassociated"*, and subsequently, SM observes that the tasks associated with that CN are also lost. This results in a set of log records with high IG, and the first region of spikes in entropy values for ***TC*** in Figure 7.6 right after *x=500*. (**S2**) **Interleaving logs**. In addition, due to the interleaving of logs in the distributed system, other components in the system still continue to generate normal logs. This is manifested in the entropy drop after the initial spike. (**S3**) **Recovery attempts.** Happens when SM makes several unsuccessful attempts to *recover* from the failed state and reconnect to the lost CN. This is manifested as the second high entropy region in Figure 7.6 that ends just before *x=750*. (**S4**) **Cleaning and back to normal**. After **S3**, SM gives up attempting to reconnect to the failed CN and continues the execution by reassigning the failed tasks to the remaining CNs. In the meanwhile, it clears the data structure allocated to the failed CN. It should be noted that the outlined stages can manifest differently depending on the applications. For example, at *info* level, TS generates far fewer log lines (0.57 MBs) than TC (313 MBs). As such, TS observes less interleaving of logs compared to TC, and **S1** and **S3** manifest as one spike region in the logs.

**Finding 4.** *A compute node failure with manifestation in log files would result in higher entropy values than normal entropies, and different runs show extended computation time and additional logs related to the failure. CT of CPU-intensive applications suffers more from compute node failure than I/O intensive benchmarks.*
**Implications.** *Sudden changes in the entropy values of log records can signify a system failure.*

### 7.5.2 Storage Failure

To gain insight into storage failures, we investigate the entropy changes of HDFS logs. All nodes within the HDFS file system (*i.e.*, name node and data nodes) generate logs. We perform the experimentation for all the benchmarks and review the logs from all the nodes and measure the entropy changes as the failure happens. Due to the limited space, we focus on the entropy value changes of WordCount and DFSIO as they are I/O intensive benchmarks, and they make the most use of HDFS compared to other benchmarks. Figure 7.8 shows the entropy values over time for the name node (WC_NN), and three data nodes (WC_DNx) as the failure happens. When DN4 fails at log window 69, we observe a delayed manifestation of entropy changes in other data nodes (DN2 and DN3) which starts at *x=100*. We observe that DN2 and DN3 directly contact DN4 (which is not available) to retrieve some blocks of data, and hence this results in failure log messages and hence higher entropies. By default, DNs are configured to send heartbeat signals to NN every 3 seconds. However, in case of a DN failure, NN marks an unresponsive DN dead after 10mm:30ss[1], which at that time manifests in high entropy values in NN logs. The log windows in Figure 7.8 show the entire execution span for WC, which on average finishes within four minutes, and hence, we do not observe entropy value changes in NN in the plotted timeline. Similarly, Figure 7.9 represents the entropies for DFSIO, another I/O intensive task in our benchmark set when a data node (DN4) fails. We have also shown the entropy values for DN2 and DN3 during a normal run for comparison. The peaks show failure log messages with some normal interleaving logs. Failure for DN4 happens very close to the start of the x-axis and thus the initial peaks for DN2 and DN3.

**Finding 5.** *We observe noticeable entropy changes in the HDFS logs of I/O intensive benchmarks as the storage failure occurs. CPU-intensive benchmarks that have minimal interaction with HDFS do not generate enough HDFS logs for a meaningful log analysis.*
**Finding 6.** *I/O intensive applications that read and write large volumes of data to the distributed storage will be negatively affected the most as a result of a storage node failure, whereas CPU-intensive applications (i.e., with minimal R/W to the storage) will be less impacted. Thus, CT of I/O tasks becomes prolonged due to the storage failure and the application's log size is also partially increased as it captures extra failure log records.*
**Implications.** *As failures are manifested as higher information gain, entropy-based anomaly detection approaches can be applied for online log analysis to isolate the higher entropy regions and further investigate the failures.*

[1] https://issues.apache.org/jira/browse/HDFS-3703

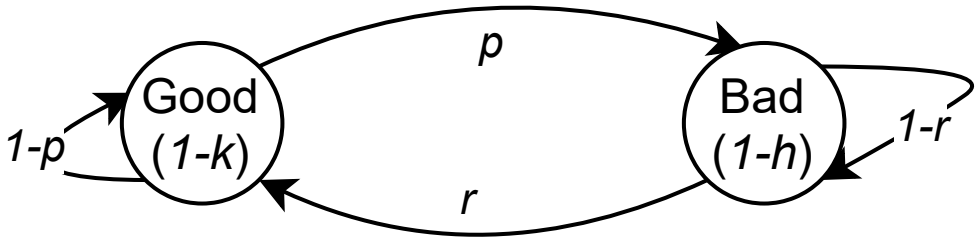



Figure 7.7: Gilbert-Elliot communication interference model.

### 7.5.3 Communication Interference Modeling

If the network connection between distributed nodes permanently disconnects, the observable failure outcome would be similar to the permanent failure of the compute/storage node, as that node becomes unreachable from the cluster manager. In contrast to permanent failures in Sections 7.5.1-7.5.2, we here investigate intermittent network interference, *e.g.*, packet loss, to gain insight into non-permanent failures which are manifested as ***performance degradation***. To emulate a realistic network traffic model, we implement Gilbert-Elliot capacity modeling approach [231, 143], which is comprised of *Good* and *Bad* states (Figure 7.7). This model offers a more realistic emulation for network impairments, rather than simple packet loss.

An example usage configuration for Gilbert-Elliot scheme would be as follows: ‘‘`tc qdisc add dev dev_name root netem loss gemodel 2% 15% 30% 1%.`’’ In this example, the error rate in Good *(1-k)* and Bad *(1-h)* states are *1%* and *30%*, respectively, and the probability of transitioning to Good *(r)* and Bad *(p)* states are *2%* and *15%*, respectively. In the following experiments, we vary the error rate in Bad state, *i.e., (1-h)*, in the range of (0%, 45%) and measure the computation time. In addition, we also evaluate the *‘combined failure’* (Case **4** in Section 7.5) by disconnecting one of the machines in the cluster that hosts both compute and data nodes.

**Heterogeneous cluster.** For the purpose of experimentation, we also define a new configuration that has three slave/data nodes but one node is smaller as it is using half of the cores and memory (*i.e.*, 12 cores and 16 GBs of memory instead of 24 cores and 32 GBs of memory). This is in contrast to the homogeneous cluster (Figure 7.3) that all the three slave/data nodes are the same size (24 cores and 32 GBs of memory). The rationale to include the heterogeneous configuration is to compare it with the performance degradation scenario that appears as a result of network interference.

**Results.** Figures 7.10a and 7.10b show the evaluation for **communication interference** and **combined failure** for WordCount and TransitiveClosure as examples of

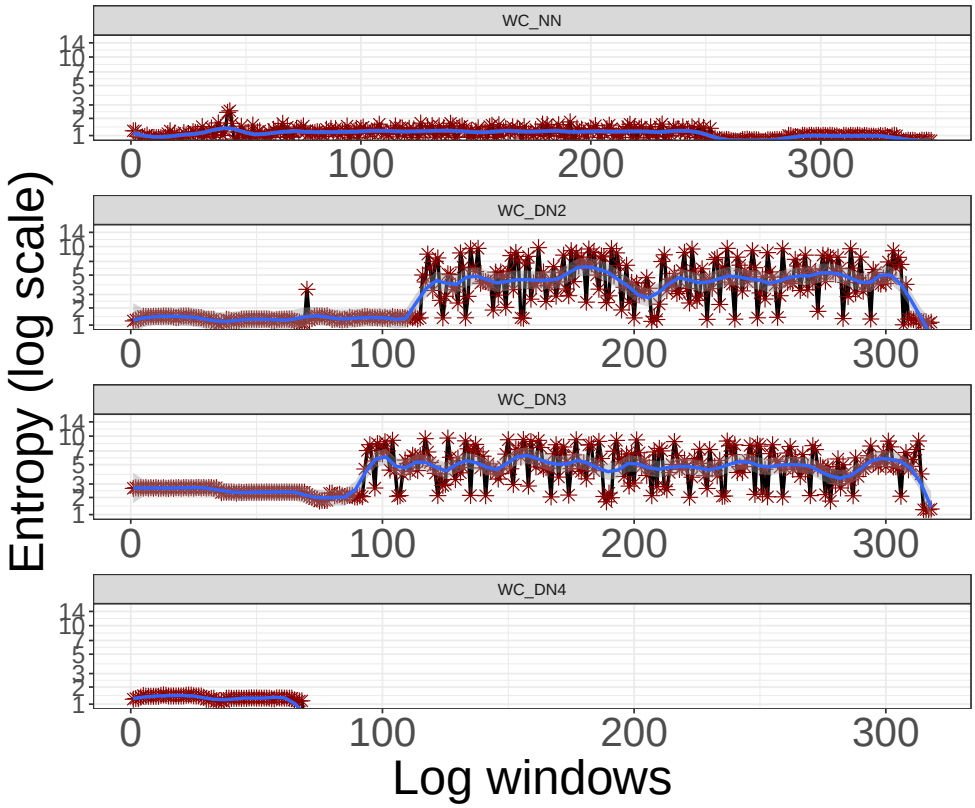



Figure 7.8: Entropy values for log windows for WordCount with HDFS's data node failure.

I/O intensive, and iterative and CPU-intensive benchmarks, respectively. We refer to the graphs by their labels in the figures, *i.e.*, (A)-(E). Graph (A) shows the computation time as a result of a combined failure for a cluster with 2 nodes, *i.e.*, two compute nodes and two storage nodes after a machine that hosts both compute and storage nodes fails. Graph (B) shows a *homogeneous* cluster with three nodes, and Graph (C) shows a *heterogeneous* cluster in which the third node is smaller ('3Nodes-1small'). No network interference is applied to (A), (B), and (C). Graphs (D) and (E) are equivalent to (B) and (C), respectively, but with added network interference. In Figure 7.10, as the communication interference increases, the CT time for (D) and (E) increases, and for values higher than 15% for WC and 10% for TC, the computation time of a cluster with interference surpasses a cluster with combined failure, Graph (A), *i.e.*, two nodes in the cluster. We also observe that I/O intensive benchmarks that require to transfer a large amount of data among the nodes in the cluster suffer more than CPU-intensive benchmarks that use the network to a lesser degree.

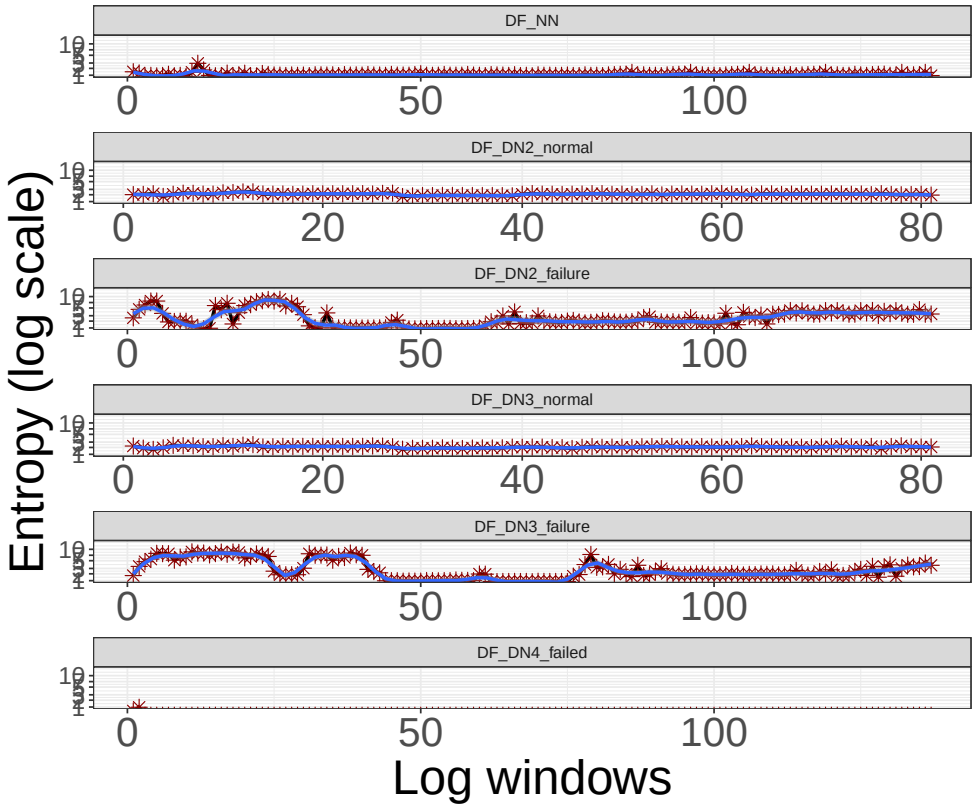
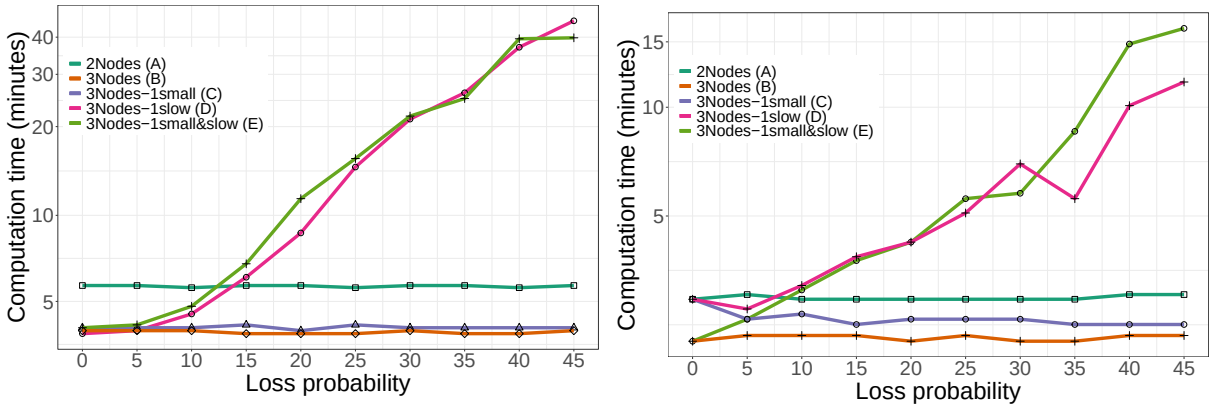



Figure 7.9: Entropy values for log windows for DFSIO with HDFS's data node failure. We have also shown the entropy values for DN2 and DN3 during a normal run for comparison. The peaks show failure log messages with some normal interleaving logs. Failure for DN4 happens very close to the start of the x-axis and thus the initial peaks for DN2 and DN3.

**Finding 7.** *As the communication interference increases, the computation time increases, and communication interference is manifested as a performance degradation and not a complete failure.*
**Finding 8.** *When the interference increases beyond a certain threshold, the negative impact of the performance degradation surpasses the impact of a complete failure because, for each stage of the computation, the faster nodes are awaiting the completion of the slow node.*
**Implications.** *Distributed scheduling algorithms that can detect slow nodes in the system and remove them from the computation can benefit the entire system's performance.*

**Entropy values.** Figure 7.11 shows the effect of drop rate in Bad state *(1-h)* for




(a) WordCount (b) TransitiveClosure

Figure 7.10: Execution time for WC and TC during the communication interference and combined failure.

WordCount logs. The top-left graph ('w/o') shows the entropy of Spark's logs with zero communication interference, and we gradually increase the drop rate from 5% to 45% (in steps of 5%) and plot the entropy values from the start to the end of the execution for selected percentages, *i.e.*, (20, 25, 30, 35, and 45)%. Our observation is that due to the non-deterministic nature of network interference, performance degradation is indirectly manifested in the logs and their corresponding entropy values, in contrast to having clear regions with high entropies (Figures 7.6-7.8). We also observe that entropy values start to climb as the interference percentage increases. In addition, higher entropy values are manifested with a delay towards the end of the execution as the system experiences timeouts and aims to reestablish the connection with the unstable node or resubmit the failing tasks to other nodes in the distributed system. Therefore, we hypothesize that a combination of execution log records and system metrics, such as average task completion time for speculative execution (Section 7.5.4), are required to identify performance degradation cases [162].

**Finding 9.** *As the communication interference increases, the entropy values gradually increase with a delay.*
**Implications.** *Failures that manifest as a gradual system slowdown and performance degradation are harder to detect than complete failures solely with logs. As such, other system metrics, e.g., average task completion time, can be applied in conjugation with logs.*

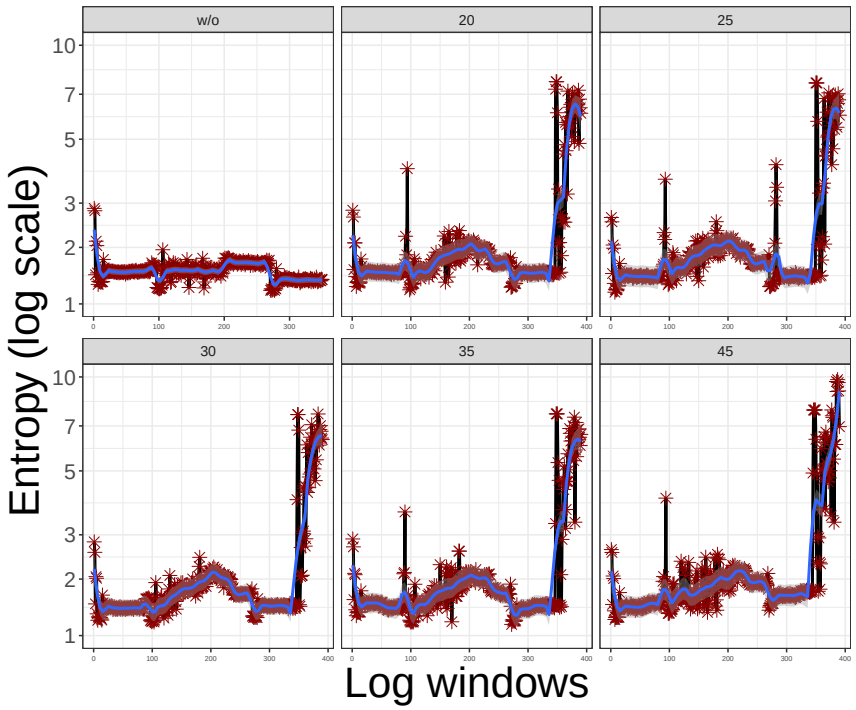
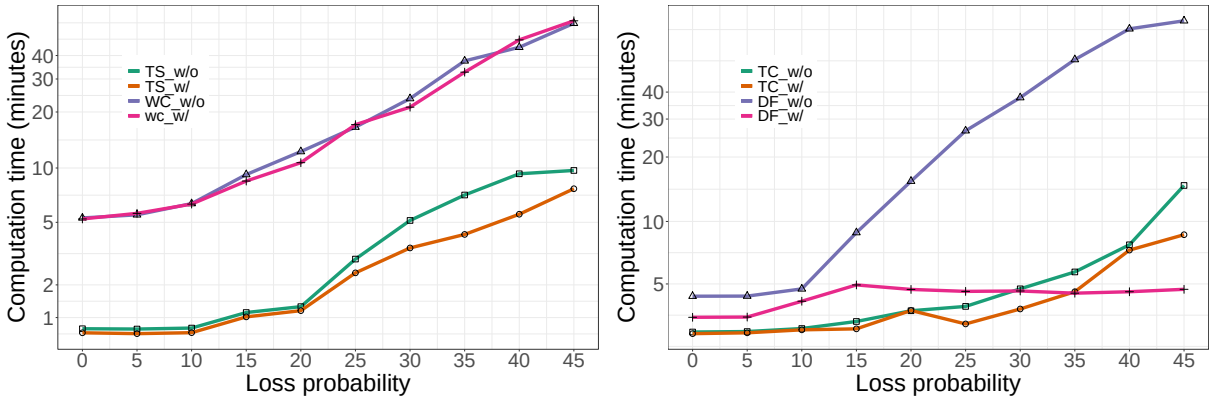



Figure 7.11: Entropy values for log windows for WordCount for different values of drop rate *(1-h)*.

### 7.5.4 Discussion

**Speculative execution.** Findings 7-9 imply that since communication interference is manifested as intermittent failures, as opposed to a complete compute or data node failure, they are harder to detect and diagnose from the log files. Therefore, we suggested the usage of distributed scheduling algorithms that can detect the slow nodes and utilization of system metrics (*e.g.*, average task's computation time) in conjunction with logs for more effective problem diagnosis. Spark provides a feature known as speculative execution (`spark.speculation`) [13] that if enabled, allows resubmitting slow tasks to other nodes in parallel, and proceeds as soon as any of the task instances completes its execution. Figure 7.12 shows the non-speculative ('w/o') and speculative ('w/') runs for WC, TS, TC, and DF. For WC, although the computation is moved to another node, because large amounts of data are being shuffled through the network interference to the new node, speculation would not show a noticeable improvement. For DF, we observe as the network interference in-




(a) WC and TS

(b) TC and DF

Figure 7.12: Speculative execution for different benchmarks.

creases, without speculative execution, the write pipeline[2] fails more often, and this results in a noticeable increase in the execution time of this benchmark. With speculative execution, the slow tasks are moved to other nodes, hence with a different write pipeline, which significantly helps with containing the network failures. In short, the main benefit from speculative execution in the distributed environment comes from moving tasks from the faulty node with network interference to other nodes, which helps to avoid task re-execution and data re-transfer due to the network uncertainty. Our observation is that speculative execution only marginally improves the execution time, and in general, after a certain level, even with speculative execution enabled, the distributed systems performance becomes slower than removing the node with lower performance completely from the cluster.

**Cluster heterogeneity.** We experimented with homogeneous and heterogeneous clusters in Section 7.5.3. Although heterogeneity by design, *i.e.*, having a machine smaller than other machines in the cluster, is well understood [245], heterogeneity that is the result of performance degradation and partial failure is left untreated. In our case, the rationale to include experiments with a heterogeneous cluster is to simulate a scenario that heterogeneity is introduced in the distributed platform because of performance degradation. In other words, although the original design is homogeneous, heterogeneity can still happen due to a variety of reasons, such as communication interference, which can negatively impact the entire system's performance. Also, one of the factors that limits Spark speculation performance is its assumption about operating in a homogeneous environment, which is not the case in a performance-impaired cluster. This would encourage further research to

[2] https://stackoverflow.com/questions/37531946/what-is-hdfs-write-pipeline

investigate possible scenarios and solutions for failure-induced heterogeneity.

**Slow distributed file system.** Although with the replication factor of three in HDFS, there is no single point of failure, partial failures can negatively affect the performance of the entire file system. In our case, the slow network connection for one of the data nodes due to an induced drop rate negatively impacts the performance of the entire HDFS. A slow data node still continues to send heartbeats successfully, and the HDFS name node will keep redirecting clients to the slow DN, and therefore, degrade the performance of the entire cluster. Although HDFS provides few settings to detect and report slow data nodes, it does not provide a mechanism to automatically bypass the slow DNs, as they are still sending heartbeat signals to the name node. Thus, we foresee further research to investigate mechanisms, similar to Spark's speculative execution, to obtain data from other available data nodes in case a data node becomes intermittently unavailable or slow.

**Implications on fault tolerance.** Our observation is that information gain is helpful in zooming in the failure regions (*i.e.*, spikes in the entropy values), which means that these regions of logs with higher IG can be isolated and quickly reviewed by users and practitioners for failure diagnosis, which then results in faster system recovery from a failure. Additionally, we emphasize that although the spark cluster is fault-tolerant by design, its performance is impaired due to the failures. Thus, we envision that quicker detection and recovery of failed compute and storage nodes directly connects with and benefits the robustness and fault-tolerance of the system. A speedy return to the normal state will allow the system to tolerate additional failures.

**Scope of our study and threats to validity.** Our experiments are conducted on a limited commodity cluster and on different categories of Spark benchmarks, as a popular and commonly-used parallel and distributed platform. To generalize our findings, further research may consider deployment of large-scale and commercial clusters for analysis, and on other common distributed platforms. Moreover, although we simulated various types of distributed failures, it is still required to obtain real-world failures and analyze the logs to confirm our findings.

## 7.6 Case study

In this section, we use logs from real abnormal scenarios for labeled OpenStack logs [107]. We first parse the logs to extract their templates and then group the logs based on their *instance_id*. *instance_id* serves as an identifier for a particular task execution sequence. Figure 7.13 shows the timeline of 52 log windows with 4 injected abnormal OpenStack

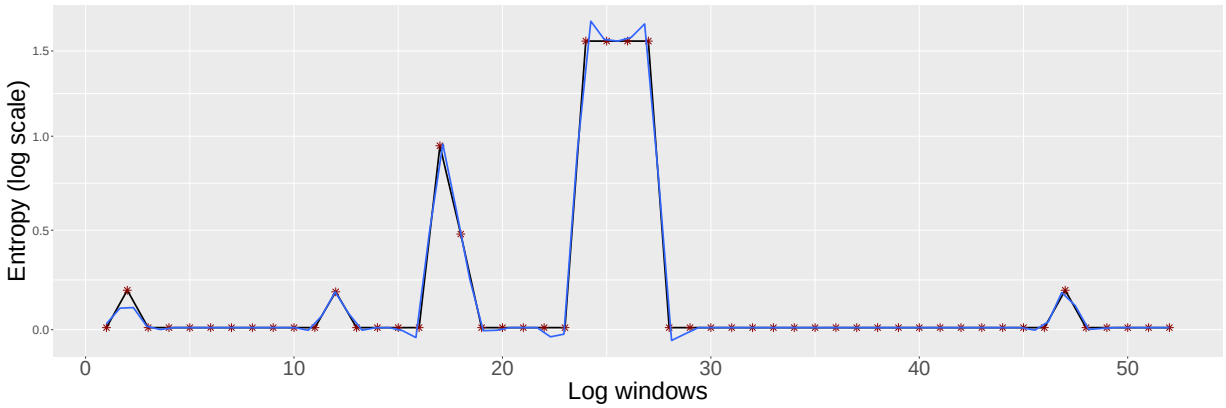



Figure 7.13: Openstack entropy values for log sequences with four anomalous VM log records.

VM sequences in the center for $x \in [24, 27]$. There is also one false positive high entropy value at $x = 17$. By leveraging the *Hampel* filter and outlier detection approach proposed in prior work [126], we can reach *F-Measure* ($\frac{2 \times precision \times recall}{precision + recall}$) and *Balanced Accuracy* ($\frac{TP}{2(TP+FN)} + \frac{TN}{2(TN+FP)}$) of 0.89 and 0.98, respectively. This signifies the effectiveness of the information gain approach that is achieved by measuring entropy values of log sequences in detecting OpenStack's abnormal scenarios.

**Computation time.** In our analysis, the training of the natural language model happens only once on normal logs. Then, testing the log records while they are generated is rather fast. We performed a quick measurement and received on average *2.4* milliseconds execution time (as a single thread executed on 2.40GHz Intel Xeon E5-2620) for a *4KB* log window. Relatively, the execution time for testing logs is faster than the rate of log generation, thus, the log records can be tested in real-time and observed for information gain and potential anomalies.

## 7.7 Related work

**Logging cost and gain.** Prior work on assessing logging cost does not quantitatively evaluate the system performance overhead contributed to logging levels. Ding et al. [104] performed a survey of logging practices among Microsoft developers and listed their findings on overheads associated with logging reported by the developers. Mizouchi et al. [229] presented PADLA, an online method to dynamically adjust the logging level, and, conse-

quently, limit the logging cost. Miranskyy et al. [228] reviewed challenges for log analysis of big-data systems, among them *limited storage* and *unscalable log analysis*. Goshal *et al.* [128] discussed the provenance, *i.e.*, the origin, of logs and how it correlates with log levels and types of applications in large-scale systems. Different from the prior research, our study aims to quantitatively analyze the costs and benefits associated with the level of logging and the information gain.

**Spark's performance evaluation.** Spark [327] has been introduced for massively parallel data analytics, which improves on its predecessor, Hadoop MapReduce [99], by utilizing in-memory storage of intermediate results for iterative applications, and bringing more flexibility in performance and the programming model [283, 254]. Mavridis and Karatza [222] evaluated various log file analyses with the cloud computational frameworks, Apache Hadoop and Apache Spark, and, experimentally, Spark achieved the best performance. Lu *et al.* [214] performed log-based anomaly detection for Spark. Our study is different as we leverage Spark's logs to evaluate the cost and IG associated with log levels.

**Information gain and NLP.** In this research, we propose a new perspective along with validated metrics to evaluate the impact of different verbosity levels on cost and information gain from log statements with natural language processing (NLP) approaches. Compared with related research [54, 193, 123], our approach is orthogonal to such efforts that aim to suggest the proper logging statement or its VL by extracting features from the source code. These works rely on the existing logging statements to suggest logs for newly composed instances of code. However, we aim to bring attention to the trade-offs between logging cost and the information gain, and that failures are manifested as higher information gain in logs.

## 7.8 Closing Remarks

The goal of our work is to provide a quantitative assessment of logging cost in different verbosity levels and how that translates to information gain in distributed systems. Therefore, we evaluate the impact of log verbosity levels on performance and storage overhead, and the information gain from logs for various Spark Benchmarks. We also experiment with synthesizing various categories of distributed failures for compute and data nodes, and network interference, and measure the effect of failures in execution time, the volume of the generated logs, and the information gain. Lastly, we provide a case study of the application of our approach on OpenStack real failure logs. Our findings are helpful for developers and practitioners to better evaluate the costs and benefits of logging when choosing different verbosity levels and how failures can be tracked down with IG approaches. As future work,

we will look into evaluating logs of other distributed software systems and investigate how IG can be translated to more effective troubleshooting by leveraging the execution logs and system metrics.

# Part V
# Logging Cost and Gain Analysis

# Chapter 8

# On the Naturalness and Localness of Software Logs

**Abstract-** Logs are an essential part of the development and maintenance of large and complex software systems as they contain rich information pertaining to the dynamic content and state of the system. As such, developers and practitioners rely heavily on the logs to monitor their systems. In parallel, the increasing volume and scale of the logs, due to the growing complexity of modern software systems, renders the traditional way of manual log inspection insurmountable. Consequently, to handle large volumes of logs efficiently and effectively, various prior research aims to automate the analysis of log files. Thus, in this chapter, we begin with the hypothesis that log files are natural and local and these attributes can be applied for automating log analysis tasks. We guide our research with six research questions with regards to the naturalness and localness of the log files, and present a case study on anomaly detection and introduce a tool for anomaly detection, called *ANALOG*, to demonstrate how our new findings facilitate the automated analysis of logs.

**Keywords:**
software systems, logging statements, log files, entropy, natural language processing, naturalness, localness, NLP, anomaly detection

**An earlier version of this chapter has been published in IEEE/ACM Conference on Mining Software Repositories [126].**

## 8.1 Introduction

Logging is an everyday programming practice and of great importance in modern software development, as software logs are widely used in various software maintenance tasks. Based on its granularity, logging allows developers and practitioners to investigate the inner-workings of software systems, and track down problems as they arise. Because of the rich information that resides in logs and the pervasiveness of logging, logs enable a wide range of tasks such as system provisioning, debugging, management, maintenance, and troubleshooting. Examples of prior research threads associated with logs include analyzing user statistics [187], identifying performance anomalies [85, 234], diagnosing system errors and crashes [323, 314], and ensuring application security [244]. Fig. 8.1 illustrates an example of a logging statement and its end product written to a log file[1].

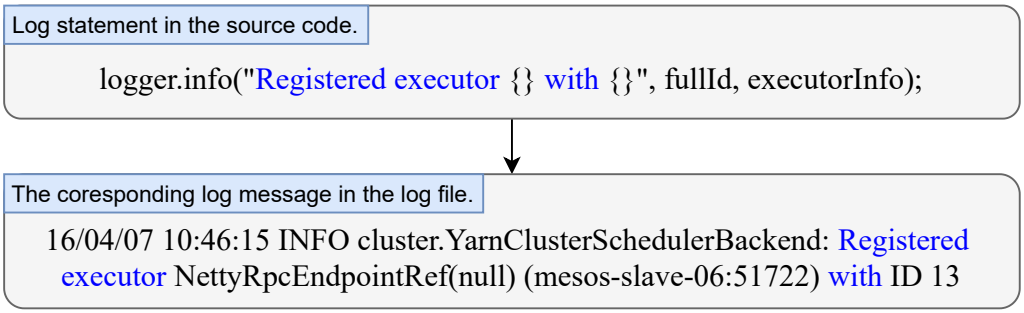



Figure 8.1: A log example from an Apache Spark application.

A Log statement is commonly supplemented with a verbosity level (*e.g.*, error/debug/info), a constant part (*e.g.*, *"Registered executor with"* in Fig. 8.1; also called *'log statement description'*), and a variable part (*e.g.*, *fullId* and *executorInfo* in Fig. 8.1). Additionally, logging libraries and wrappers such as Log4j [15] accompany the log statement with extra information as it is written to the log file, such as its timestamp and the component generating the log. Due to the free-form text format of the log messages written by developers, it is often an involved task to extract meaning from the log messages. Furthermore, the dynamic nature of variables, which may yield a different output in each iteration of the program, brings additional irregularity and adds to the complexity. In a parallel angle, due to the sheer volume of logs that modern software systems routinely produce, in the scale of tens of gigabytes of data per hour for a commercial cloud application [347, 226, 151], it is unfeasible to inspect log messages for crucial diagnostic information with traditional methods such as manual checks or searches with search and

[1]We use *logs* and *log files* interchangeability.

`grep` scripts. As such, although tremendous system diagnosis and maintenance advantage is beclouded in the logs, how to effectively and efficiently analyze the logs remains a great challenge, and subsequently, automatic log analysis tools and approaches are highly sought after [347].

To fill in this gap and pave the way towards the goal of automated log analysis, prior research has proposed multiple avenues of work to automate analysis, which relies on finding patterns in logs. The analysis usually starts with log parsing [347], which aims to extract structured templates from unstructured log data. Following this step, prior research has applied a wide variety of pattern mining and machine learning approaches, to name a few, PCA-based dimension reduction [242], execution path and variable value inference [323], learning model [234], event correlation graphs [118], and temporal correlation mining [103].

Despite the presence of prior log analysis approaches, we believe prior research has not fully utilized the natural language attributes of software systems [155, 298] for log file analysis. Therefore, in this study, we focus on the natural language processing (NLP) characteristics of log files, and our main hypothesis is:

> *Logs, as an artifact of software systems, similar to programming languages and source code, are natural and locally repetitive and predictable. Thus, natural language models can capture these attributes and leverage them for automated analysis of log files.*

Prior research such as [155] and [298] have shown that software's source code, similar to natural language ($\mathcal{NL}$) corpus, is repetitive, predictive, and local. As such, statistical language models yield promising results once applied in software engineering tasks. Authors in [155] and later in [298] showed that n-gram language models perform well in the source code's modeling and leveraged this fact to propose a cache language model for code suggestion. Most recently, He *et al.* [145] showed that logging descriptions within the source code also follow the natural language repetitiveness observed in the software systems. In this study, inspired by the prior work, we investigate if log files possess NLP characteristics and how to leverage this feature for automated analysis of logs.

There are several findings and implications stemming from our study. We observed that log files show a high degree of repetitiveness and regularity (Finding 1), and the regularity is project-dependent (Finding 2). Zipf's law shows the high-rank tokens happen more often in the logs than in English text (Finding 3). Findings 4 and 5 shed light on the log localness and confirm that logs are *endemic* and *specific*. At last, our NLP-based anomaly detection approach achieves high F-Measure and Balance accuracy scores, which

illustrates an application of NLP in log analysis (Finding 6). In summary, our study makes the following contributions:

- We conduct the first empirical study on the utilization of natural language for log files by evaluating eight system logs and two English corpora. We have provided our dataset to encourage and facilitate further research [5].
- We demonstrate the naturalness and localness of logs through several research questions (RQs) by utilizing n-gram models and self- and cross-project entropy calculations.
- We introduce *ANALOG*, an NLP-based log file anomaly detector, to illustrate the potential benefits of NLP in log analysis.

We organize the rest of this chapter as follows. Section 8.2 reviews the background and motivation. Section 8.3 presents our research questions (RQs), and we quantitatively analyze the naturalness and localness of logs in Sections 8.4 and 8.5. Section 8.6 demonstrates the use of language models (LMs) in a case study for anomaly detection. Related work and threats to validity are in Sections 8.7 and 8.8. Finally, we conclude the study with some avenues for future work in Section 8.9.

## 8.2 Background and Motivation

Natural language processing (NLP) models (*e.g.*, the n-gram model [9]) are statistical models that *estimate* and *evaluate* the probability of a sequence of words or tokens. During *estimation*, the model assigns a probability to sequences of words (or tokens) with *maximum likelihood estimation* (MLE). During *evaluation*, the model predicts the probability of whether the sequence under test belongs to the training corpus. The predictable and repetitive characteristics of common English corpora, which statistical NLP techniques extract and model, have been the driving force of various successful tasks, such as speech recognition [59] and machine translation [219]. Subsequently, software engineering researchers [155, 298, 51] have shown that software systems are even more predictable and repetitive than common English text, and language models perform better on software engineering tasks than English text. As such, tasks such as code completion [262] and code suggestion [66] utilize n-gram models for their predictions. Recently, He *et al.* [145] showed that log statements' descriptions (LSDs) within the source code (Fig. 8.1) also follow natural language characteristics. Because LSDs in the source code are considered in

isolation and they do not completely determine what is in the log files, which is an arbitrary sequence (*i.e.*, not isolated) of log prints with LSDs and, additionally, the dynamic runtime value of the variables included by virtue of the execution of log statements, the naturalness of LSDs in the source code does not guarantee the naturalness of log files. Thus, our goal is to validate the NLP attributes of logs and foster research and employment of NLP methods for automated log analysis.

**N-gram language models.** Formally, considering a sequence of tokens in the corpus under consideration (in our case, log files), $S = a_1, a_2, ..., a_N$, the n-gram model statistically estimates how likely a token is to follow preceding tokens. Thus, the probability of the sequence is estimated based on the product of a series of conditional probabilities [155]:

$$P_\theta(S) = P_\theta(a_1)P_\theta(a_2|a_1)P_\theta(a_3|a_1a_2)....P_\theta(a_N|a_1...a_{N-1}) \tag{8.1}$$

which is equal to:

$$P_\theta(S) = P_\theta(a_1).\prod_{i=2}^{N} P_\theta(a_i|a_{i-1}, a_{i-2}, ..., a_1) \tag{8.2}$$

where $a_1$ to $a_N$ are tokens of the sequence S and the distribution of $\theta$ is estimated from the training set with MLE[2]. The metric to assess the performance of an n-gram model $\mathcal{M}$ is *perplexity* ($PP$), which is the inverse probability of the test sequence:

$$PP_\mathcal{M}(S) = \sqrt[N]{\frac{1}{P(a_1).\prod_{i=2}^{N} P(a_i|a_{i-1}, a_{i-2}, ..., a_1)}} \tag{8.3}$$

For model $\mathcal{M}$'s performance evaluation, perplexity and its log-transformed version, *cross-entropy*[3], $H_\mathcal{M}$ [4] are often used interchangeably: $H_\mathcal{M}(S) = \log_2 PP_\mathcal{M}(S)$. Equation 8.3 explains that the higher the probability of a sequence, the lower the $PP$ value will be. In other words, the lower the *perplexity* (and likewise *cross-entropy*), the less surprising the new token sequence is for the model $\mathcal{M}$, and hence the higher probability that the token sequence under investigation belongs to the same corpus. Thus, an appropriately trained n-gram model will predict with low probabilities (and high *perplexity* values) if it deems that the content of a test sequence does not belong to the corpus used for training.

**Motivation.** Encouraged by the prior work in the utilization of NLP for software engineering systems, in this work, we investigate the natural language characteristics of log files. More specifically, we aim to answer the question of *"are log files natural and local?"*. Intuitively, if there are discernible repetitiveness and predictability in logs, a suitably trained

[2]For simplicity, we drop the $\theta$ in the notation onward.
[3]Often, simply called *entropy*.

language model should yield acceptable performance in distinguishing the log messages belonging to a specific system's log from the ones which look unfamiliar. We speculate this feature will benefit automated analysis of logs, and we aim to use it for software engineering tasks such as anomaly detection since anomalous log messages generally look different from normal logs. For this purpose, we exploit the n-gram language models, measure the perplexity and entropy values for different logs, and analyze our findings from the RQs.

## 8.3 Natural Language Processing for Logs

To investigate whether log files are natural and local, we analyze both natural English corpora and a collection of logs from various systems available online and applied by prior research [151, 319]. We guide our research with the following research questions (RQs) for the **naturalness** of logs:

- ***RQ1:*** *does a natural repetitiveness and regularity exist in log files?*
- ***RQ2:*** *is the regularity that the statistical language models capture merely log-nature specific, or is it also project-specific?*
- ***RQ3:*** *how does Zipf's law capture the repetitiveness of high-rank tokens in log files?*

Next, for **localness** of logs, we investigate:

- ***RQ4:*** *are log n-grams endemic to their projects*[4] *?*
- ***RQ5:*** *are log n-grams specific to their projects?*

Finally, we provide a case study of the applications of NLP attributes of logs in the final RQ:

- ***RQ6:*** *how the logs' NLP characteristics*[5] *can help with automated analysis of log files?*

[4] We use *project* and *system* interchangeably, as software systems are commonly referred to as projects during their developments, *e.g.*, Apache Hadoop project.

[5] We use *'natural language characteristics or NLP characteristics'* to cover both of *naturalness* and *locallness*.

Clarifying the above research questions with quantitative analysis for logs and English corpora enables us to find supporting evidence on our hypothesis on the naturalness and localness of logs, which we later use for anomaly detection. We continue with the description of the dataset that we used in our analyses.

**Data description.** We utilize the data publicly available by prior research [319, 151] as the base for our analyses. In summary, we select eight log files from a wide range of computing systems and two English corpora. We briefly review each of the analyzed logs in the following.

❶ **HDFS.** The HDFS [23] log is generated from a Hadoop cluster. HDFS is a distributed and resilient-to-failure file system for commodity servers, which brings in reliability. Prior research [347, 319] has used this data set for log parsing and compression.

❷ **Spark.** Apache Spark [16] is a popular and efficient big-data processing framework. This log data is aggregated from event logs of a spark cluster of 32 machines. Prior research has utilized this data for log message pattern extraction [150, 347].

❸ **Firewall.** Firewall log is the collection of firewall process activities during the network operation of operating systems, such as Windows's firewall process. Prior work has used this type of data for frequent pattern extraction and compression [319, 144].

❹ **Windows.** The Windows log belongs to Windows 7's component-based servicing (CBS) initially stored at *C:/Windows/Logs/CBS*, which collects logs on package and driver installations and updates. It is used for log parsing and anomaly detection in prior studies [347, 151].

❺ **Linux Syslog.** The Linux Syslog is the standard logging system in Linux which collects logs from various concurrently running applications on a single machine, such as networking and *cron* logs, and Linux kernel logs. Syslog is used to analyze compression on log data in prior research [319].

❻ **Thunderbird.** The Thunderbird dataset is collected from a Linux supercomputer cluster from Sandia National Labs (SNL), which is the aggregation of Syslogs from different machines. This dataset is used in prior work for log compression and log analysis [241, 319].

❼ **Liberty.** The Liberty log is an aggregated dataset of Syslog-based events on a supercomputer system, which was examined for log analysis and alert detection in previous studies [241, 243].

❽ **Spirit.** Similar to Thunderbird and Liberty, Spirit is also a supercomputer log data from the Spirit supercomputer system, used for gaining insight on failure diagnosis of large-scale systems [241].

Besides the log data, we also analyze two natural language corpora:

❶ **Gutenberg.** Project Gutenberg corpus [10] is a collection of over 60,000 English books [11]. The Gutenberg corpus has been studied to examine the performance of natural language tools [67], and NLP evaluation of software systems [145, 155].

❷ **Wiki.** The Wikipedia corpus is the data collected from English articles of Wikipedia. This data has been previously applied to evaluate the performance of different text compressors [120].

| **Logs** | **Category** | **Lines** | **Tokens** | |
|---|---|---|---|---|
| | | | Total | Unique |
| HDFS (HS) | Distributed system | 7527525 | 259293940 | 875434 |
| Spark (SP) | Distributed system | 13221789 | 308093706 | 673725 |
| Firewall (FW) | Operating system task | 4763441 | 418012413 | 2655410 |
| Windows (WD) | Operating system | 4408109 | 338718867 | 37023 |
| LinuxSyslog (LS) | Operating system | 5315580 | 348036971 | 454755 |
| Thunderbird (TB) | Supercomputer | 6018428 | 357613165 | 102699 |
| Liberty (LB) | Supercomputer | 7316856 | 352384327 | 216946 |
| Spirit (ST) | Supercomputer | 7983346 | 366526309 | 320426 |
| **English corpora** | **Description** | **Lines** | **Tokens** | |
| | | | Total | Unique |
| Gutenberg | English books | 20867266 | 232266714 | 370164 |
| Wiki | English articles | 5815221 | 173380056 | 3963045 |

Table 8.1: System logs and English corpora statistics.

Table 8.1 summarizes the system logs and the English corpora that we studied. We categorize logs based on their domains such as *distributed system, operating system (OS), OS task, and supercomputer*. In order to make a fair comparison among different log files and between log files and natural language corpora, all the files are curtailed at the same size of 1 GB. We show the number of lines, calculated using Unix *"wc -l"* on each file,

and the number of tokens. For tokens, we have listed the total number of tokens, and the unique tokens count for each file.

**Data preprocessing.** Prior to the application of NLP models, we require performing some pre-processing steps on the raw log data and English corpora. In order to be case-insensitive, we initially normalize all the letters to lower cases. We then tokenize all the data to extract words and special symbols. For n-gram model training and testing, we use Kenlm [152, 153] library. For different analyses that follow later in this study, we create n-gram models for different sizes of n in the range of $n \in (1, 10)$ for each data file, *i.e.*, for unigrams (single tokens), bigrams (pairs of adjacent tokens), and so on.

## 8.4 Naturalness of Logs

In this section, we investigate the naturalness of log files guided with a set of RQs which follows.

### 8.4.1 RQ1: does a natural repetitiveness and regularity exist in log files?

Prior research has leveraged the natural predictability of n-grams in English corpora in applications, such as speech and handwriting recognition [59, 255], and machine translation [219]. This observation is also noteworthy for artifacts of software systems, such as user documents, repositories, and logs. It is beneficial to explore whether similar repetitiveness and predictability of n-grams exist in log data as it benefits the automated analysis of logs, and it is the focus of our research.

We showed in Equation 8.2 the probability estimated by the n-gram model $\mathcal{M}$ for sequence $S = a_1, a_2, ..., a_N$. For simplifying the computation, n-gram models often assume a *Markov* property, which states that for a language model of order $n$, token occurrences are approximated only by the $n-1$ tokens that precede the token under consideration [168]. For example, for a 5-gram model, the probability of $a_i$ appearing after the sequence of $a_1, a_2, ..., a_{i-1}$ is approximated by the probability of the four prior tokens:

$$P(a_i|a_1...a_{i-1}) \cong P(a_i|a_{i-4}a_{i-3}a_{i-2}a_{i-1}) \tag{8.4}$$

To calculate the probabilities, the NLP model is estimated on a training set using the maximum likelihood-based frequency-counting of token sequences. Therefore, we estimate

the probability of $a_i$, $i_{th}$ element in Sequence S, which follows tokens $a_{i-1}, a_{i-2}, ..., a_{i-n+1}$ and order n model with:

$$p(a_i|a_{i-1}a_{i-2}...a_{i-n+1}) = \frac{count(a_i a_{i-1}a_{i-2}...a_{i-n+1})}{count(a_{i-1}a_{i-2}...a_{i-n+1})} \tag{8.5}$$

Based on this estimation, the *cross-entropy* (or *entropy*, used interchangeably), $H_{\mathcal{M}}$, evaluated for model $\mathcal{M}$ for a sequence of $n$ tokens is:

$$H_{\mathcal{M}} = -\frac{1}{N}\sum_{n=1}^{N} \log_2 p(a_i|a_{i-1}a_{i-2}...a_{i-n+1}) \tag{8.6}$$

A subtle detail worth mentioning about the n-gram model is that, in practice, the n-gram model often encounters some unseen sequences during prediction. This results in the probability $p(a_i|a_{i-1}a_{i-2}...a_{i-n+1}) = 0$, and undefined values for $H_{\mathcal{M}}$. Smoothing is a technique that handles such cases and assigns reasonable probabilities to unseen n-grams. In this study, we use Modified Kneser-Ney Smoothing [179] available with Kenlm [153], which is a standard smoothing technique and gives acceptable results for software corpora, *i.e.*, sufficient statistical rigour [155]. On the other side of the spectrum, a perfect n-gram model correctly predicts all the next tokens, *i.e.*, $p(a_i) = 1$, and therefore, $H_{\mathcal{M}} = 0$. In general, lower entropy values imply that the n-gram model is more effective in predicting the sequence of tokens and capturing the regularity and repetitiveness of the corpus.

**Experiment.** To evaluate the repetitiveness and regularity in logs, we measure cross-entropy by averaging over 10-fold cross-validation: we randomly select 10 MB from each log file, ∼59,000 lines of logs, which falls well within a ±2.5% margin of error and 95% confidence interval [180, 119]. We then organize each of the log files and English corpus to a 90%–10% train-test split at ten random locations, and train the n-gram model for different values of $n \in (1, 10)$ on the 90% train split, and then test it on the remaining 10% split by measuring the average cross-entropy with Formula 8.6.

**Result.** In Figure 8.2, boxplots display the cross-entropy results for system log files, and the blue line shows the average cross-entropy for the two English corpora. Comparing the values of entropy for log data and English corpora provides an intuitive understanding of the repetitiveness and regularity of tokens in log data and common English. The horizontal axis shows the n-gram model trained with different numbers of n and the vertical axis presets the entropy. Both the single line and boxplot manifest similar trends: as the order of n-gram increases, the entropy values decrease, which implies that using larger values of n (*viz.*, more preceding tokens in Formula 8.4) yields more accurate predictions for both log data and English Corpora. 4- or 5-gram models are the optimal choice for the studied logs considering the trade-off between the entropy and model memory usage, as cross-entropy

saturates around $n \in (4, 5)$. The English corpora entropy starts at 10.19 for 1-gram models and trails down to 8.09 for 10-gram models, compared to logs entropies median that falls just below 1.8. Thus, quantitatively, the log data manifests lesser entropies, and as such, less perplexing to predict when compared to English corpora. This observation paves the way to utilize n-gram models for automated analysis of logs. Additionally, our findings are consistent with prior research on the naturalness of software source code [155, 145].

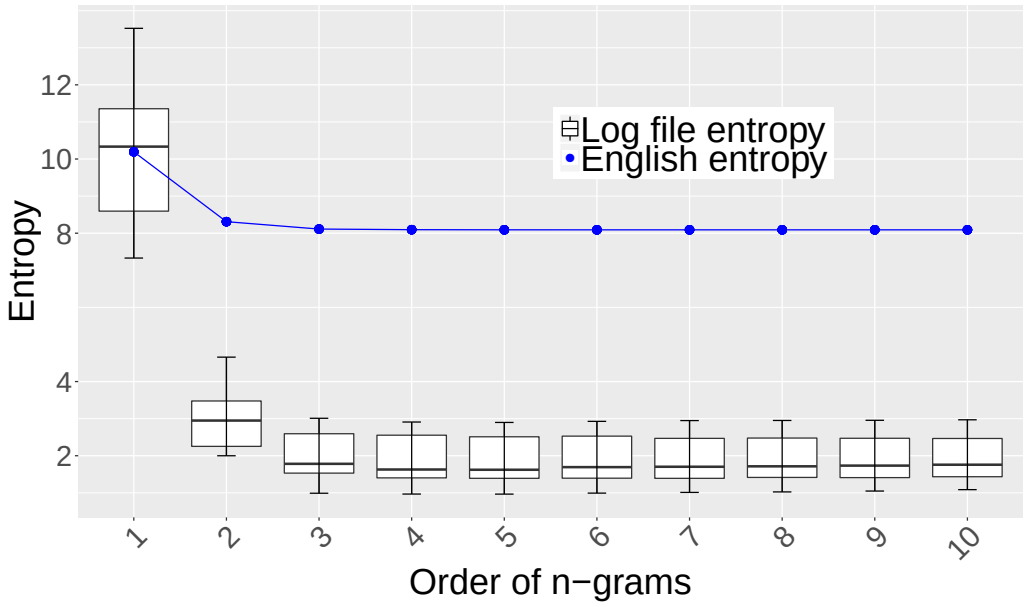



Figure 8.2: Entropy values for a sequence of n-grams for log data (boxplot) and English corpora (blue line).

### 8.4.2 RQ2: is the regularity that the statistical language model captures merely log-nature specific, or is it also project-specific?

In RQ1, we showed LMs accurately capture the repetitiveness within log files even better than English corpora. However, as we know, software systems, and subsequently, their artifacts, such as log files, have a smaller set of vocabulary when compared to the English language [155]. Thus, there exists a valid concern that the lower entropy values for logs might be the outcome of their limited vocabulary and not their regularity and repetitiveness. If this concern proves to be valid, *viz.*, if the captured regularity from the logs is solely

the result of their limited vocabulary, then we should observe similar lower entropies for cross-project evaluation. In other words, if we train the n-gram model on one log file, and then test it on another system's log, we should observe comparable entropies with inner-project entropies. As such, in RQ2, we investigate this concern by training and testing the model on different projects.

**Experiment.** In this experiment, we measure self- and cross-project entropy values by averaging over 10-fold cross-validation. We randomly sample 10 MB of data from the available 1 GB for each system and English corpora. Without loss of generality [180], we selected this sample size to make the training overhead manageable. Additionally, to confirm the results, we run some limited experiments with various sizes to confirm the results are consistent. Similar to RQ1, we split each of the samples to a 90%–10% split at ten random locations, train the n-gram a 5-gram model on the 90%, and then test it on the remaining 10%, and measured the average self entropy. As observed in RQ1, entropy values stabilize beyond 5-gram models. As such, we use 5-gram models to reach faster training and save on the memory footprint. To calculate the cross-project entropy, once we train the project on one system, we test it on all the other projects and average the values of entropies through 10-fold cross-validation, to minimize the risk of over-fitting [239].

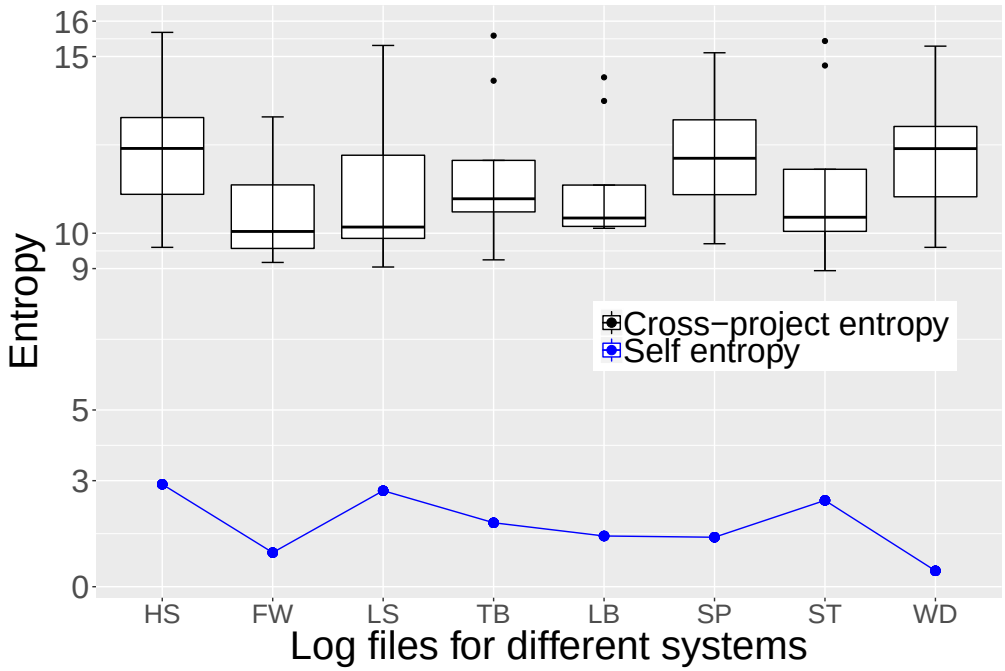



Figure 8.3: Entropy values for 5-grams cross-project versus self-project.

**Result.** Fig. 8.3 shows the self- and cross-project entropies. The x-axis lists the different logs, and for each log, the boxplot shows the range of cross-project entropies with the other seven projects. The blue line at the bottom shows the average *self entropy* of the project against itself (*i.e.*, training and testing on the same system's log). From this plot, the self-entropy values are always lower, indicating that the repetitive n-gram patterns are not the artifact of limited log vocabulary but because of the regularity and repetitiveness in each system. This regularity is different across different projects, and hence, we observe higher cross-project entropies, implying that the n-gram patterns noticeably disagree across project logs.

### 8.4.3 RQ3: how does Zipf's law capture the repetitiveness of high-rank tokens in log files?

Zipf's [348, 256] law is an empirical theorem which states that given a large sample of words in a corpus (in our case, hundreds of millions of words/tokens in Table 8.1), the probability of any word is inversely proportional to its rank in the corpus. Thus, the word rank $r$ has a probability proportional to $\frac{1}{r}$. In other words, the rank of the word ($r$) times its probability $p(r)$ is approximately a constant ($r \times p(r) \simeq const$). For the classic version of Zipf's law we have: $p(r) = \frac{\frac{1}{r}}{\sum_{n=1}^{N}(\frac{1}{N})}$, where N is the population of unique tokens. Considering that Zipf's law is not an exact value but a statistical measurement, it gives an intuitive understanding that the repetition of the high-rank tokens occupies what percentage of the document. For example, in English, the top 50 words accumulate to 35–50% of total word occurrences [331].

**Experiment & Result.** We measure the token frequency (TF) for logs and English text and then sort the tokens based on their frequencies to study Zipf's law for log files. Figure 8.4 plots the rank of tokens on the horizontal axis and the frequency counts on the vertical axis for average of Logs (red) and English text (blue) and gives us an idea of the TF distribution in a given document. Logs's TF starts higher than English and drops below as the rank increases. The top-50 tokens contribute to 70% and 51% of the entire document for Logs and English, respectively. This result is encouraging for more accurate adaptation of LMs for logs, as prior research [256] has suggested that smoothing algorithms for n-gram models, such as Good-Turing [89], and Kneser-Ney [89], could lead to better smoothing with more predictive high-rank tokens.

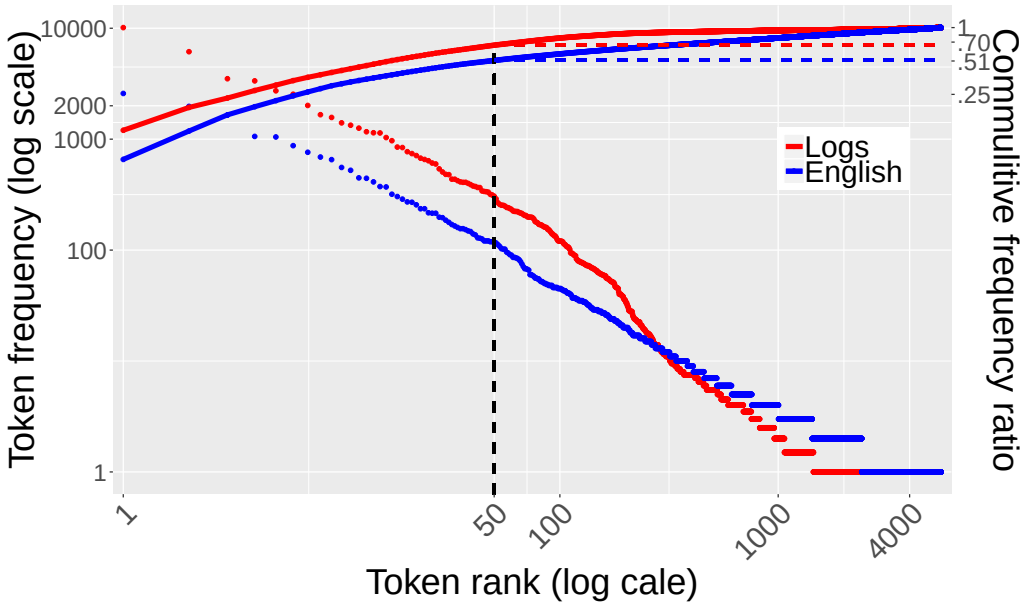



Figure 8.4: Frequency of tokens for Logs and English text.

## 8.5 Localness of Logs

In this section, we investigate the localness of log files. The localness attribute builds upon the naturalness of logs, such that besides being repetitive and predictable, logs tend to take on a specific form of repetitiveness in the *local context*. Here, a local context is a system's logs versus other systems. For this purpose, we investigate *endemism* and *specificity* of n-grams in different logs.

### 8.5.1 RQ4: are log n-grams *endemic* to their projects?

The n-grams that only appear in a single project log file (*i.e.*, *local context*) are called endemic, *i.e.*, they are endemic to that specific project's log file. In this RQ, we investigate whether there exist n-grams that are found exclusively in a local context and are endemic to a system. More specifically, we investigate what percentage of n-grams happen only in one system logs. Table 8.2 lists the average percentage of the endemic n-grams in the log files and English corpora. For both system logs and English corpora, the first row with (Freq $\geqslant$ 1) shows the percentage of endemic n-grams that appear at least once in a system, and the second row with (Freq $\geqslant$ 2) represents the percentage of endemic n-grams

that appear at least twice in a system logs. For example, 90.24% and 80.74% of 2-grams with (Freq $\geqslant$ 1) are endemic for system logs and English corpora, respectively. Similarly, 39.20% and 8.57% of 5-grams with (Freq $\geqslant$ 2) are endemic for system logs and English corpora, respectively. The percentage of the endemic n-grams for (Freq $\geqslant$ 2) generally increases for higher orders of $n$, because it becomes less likely to observe a large sequence of tokens repeatedly. For (Freq $\geqslant$ 2), we observed that for $n \geqslant 3$, because there is a very limited number of n-grams that appear more than once compared to the total number of n-grams in the same order of $n$, the percentage shrinks slightly onwards. By comparison, the percentage of the endemic n-grams in English corpora is noticeably lower than system logs. This observation validates the hypothesis that log files are *local in the context of n-grams*, even to a higher extent than natural language corpora. As such, we point out, similar to prior research [298], we can leverage this localness attribute to improve the performance of language models for logs by augmenting them with caching mechanism and store the n-grams in the local context for quick access.

| | Freq. | 1-gram | 2-gram | 3-gram | 5-gram | 8-gram | 10-gram |
|---|---|---|---|---|---|---|---|
| System logs | $\geqslant$1 | 69.09% | 90.24% | 95.44% | 97.37% | 99.60% | 99.79% |
| | $\geqslant$2 | 38.02% | 49.63% | 48.59% | 39.20% | 30.66% | 26.00% |
| English corpora | $\geqslant$1 | 56.74% | 80.74% | 93.13% | 99.63% | 99.97% | 99.98% |
| | $\geqslant$2 | 26.38% | 21.79% | 15.68% | 8.57% | 6.34% | 5.77% |

Table 8.2: System logs and English endemic n-gram stats.

### 8.5.2 RQ5: are log n-grams *specific* to their projects?

In RQ4, we experimented with endemic n-grams which happen only in one system's log. Although endemism provides insight into exclusive n-grams, we still lack insight into the overall distribution of non-endemic n-grams. This is where *specificity* comes into effect. Specificity explains whether non-endemic n-grams favor specific system logs, *viz.*, whether non-endemic n-grams happen more often in one system than another, which sheds light on the *system-wide locality* of n-grams. For example, if n-gram $\sigma$ is uniformly distributed across different log files, then all the files contain an equal number of the n-gram $\sigma$. Conversely, the more skewed the distribution becomes, the more *specific* and *localized* the n-gram $\sigma$ becomes to a *specific* log file. For example, an n-gram that happens 100 times in all the log files, if it happens 93 times in one file and only once in the rest of the files, is

highly skewed. For a set of log files $F = \{f_1, f_2, ..., f_{|F|}\}$, each non-endemic n-gram $\sigma$ can happen in more than one file with probability distribution of $p(f_i(\sigma))$:

$$p(f_i(\sigma)) = \frac{\texttt{count (n-gram } \sigma \texttt{ in } f_i\texttt{)}}{\texttt{count (n-gram } \sigma \texttt{ in Set F)}} \tag{8.7}$$

As such, to measure the skewness of the distribution of n-gram $\sigma$ in the set of log files F, defined as *locality entropy* [298, 145], $H_{\mathcal{L}}(\sigma)$, the formula is as follows:

$$H_{\mathcal{L}}(\sigma) = -\sum_{f_i \in F} p(f_i(\sigma)) \log_2 p(f_i(\sigma)) \tag{8.8}$$

Similar to *cross-entropy* values, the more skewed the distribution of the non-endemic n-grams is, the lower the *locality entropy* values will be. In the extreme case, *i.e.*, $H_{\mathcal{L}}(\sigma) == 0$, then $\sigma$ is an *endemic* n-gram, and it happens only in one system logs. $H_{\mathcal{L}}(\sigma)$ reaches its max (*i.e.*, $\log_2 |F|$) when the n-gram $\sigma$ is uniformly distributed across $F$, *i.e.*, all the files in $F$ contain equal number of n-gram $\sigma$.

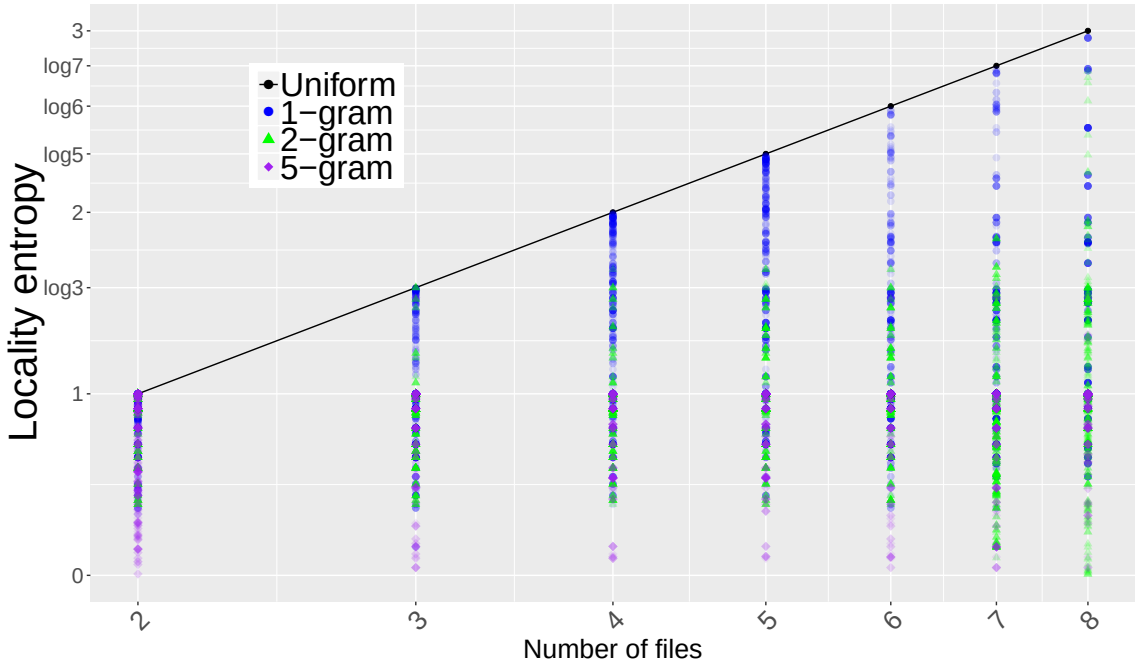



Figure 8.5: Distribution of entropies for non-endemic n-grams, grouped by the number of files. "Uniform" represents that n-grams are distributed uniformly in the files.

Fig. 8.5 shows the values of *locality entropy* for different n-gram orders for file sizes, $|F| \in (2, 8)$. The horizontal axis represents the number of files ($|F|$) that contain the non-endemic n-grams, and the vertical axis shows the *locality entropy*, $H_{\mathcal{L}}$. The n-gram with varying orders, which are marked by different colors, share similar trends. According to this figure, non-endemic n-grams' entropies commonly fall below the uniform distribution (the solid black line in the figure), and as the number of files and the order of n-grams increases, the skewness becomes more apparent, which affirms the local tendencies of n-grams.

## 8.6 RQ6: Log File Anomaly Detection

In the previous sections, we demonstrated the NLP characteristics of the log files. Based on our findings, we believe it is a valuable effort to explore how the NLP findings would help in the automated analysis of log files. As such, in the following, we focus on anomaly detection (AD) through log files by utilizing NLP. As modern large-scale computer systems continue to expand and service millions of users, their dependability becomes even more critical, and any noticeable downtime could result in millions of revenue and quality-of-service loss [151, 14, 2]. Consequently, to improve reliability, online observation of running systems for unforeseen abnormal behavior and potential problems has been the focus of prior research [68, 107, 116, 150, 207, 212, 311]. As log files contain information on the dynamic state of the system, prior research has utilized logs for AD [181, 212, 337, 332]. Log-based AD's goal is to accurately detect runtime system anomalies by processing the rich data gathered in the log files. AD is often modeled as a binary (*i.e., yes or no*) decision problem such that the input to the AD algorithm is a vector (or matrix) of events or time intervals, and the algorithm decides whether each event or interval is normal or abnormal. Encouraged by our findings from the naturalness and localness of the logs, we introduce ***ANALOG*** (**A**nomaly detection with **NA**tural language and **LOG**s) in the following.

**Approach.** We propose to utilize NLP techniques and consider log files as natural language sequences. Figure 8.6 shows the steps involved in our ANALOG approach:

- Initially, we collect the log files for the system under analysis during its normal behavior, *i.e.*, no anomalies. Similar to the approach in RQ1, we divide parts of the logs for training and testing of the n-gram model. We then preprocess and tokenize the logs and feed them into the n-gram model.
- Next, we *train* the n-gram model on the training data for an order of n which results in a good balance of performance and memory footprint. In our analysis, values in the range of $n \in (4, 5)$ are sufficient. During the training phase, we establish a ***baseline*** for

entropy values for each system. Our analysis shows that normal log sequences result in entropy values, which are *'distinguishably'* lower than entropy values for abnormal log sequences. In this step, we also establish a ***threshold (Th)*** for the maximum value of normal entropies.

- Later, during the *testing phase*, we look for anomalies as the test log data is fed into the NLP model, and if no anomaly is detected, we achieve comparable to baseline perplexity or entropy values. On the other hand, if the test log data contains abnormal sequences, which appears surprising and perplexing to the n-gram model, the entropy values will be higher than the established baseline. Then these log sequences are singled out for further analysis by practitioners or developers.

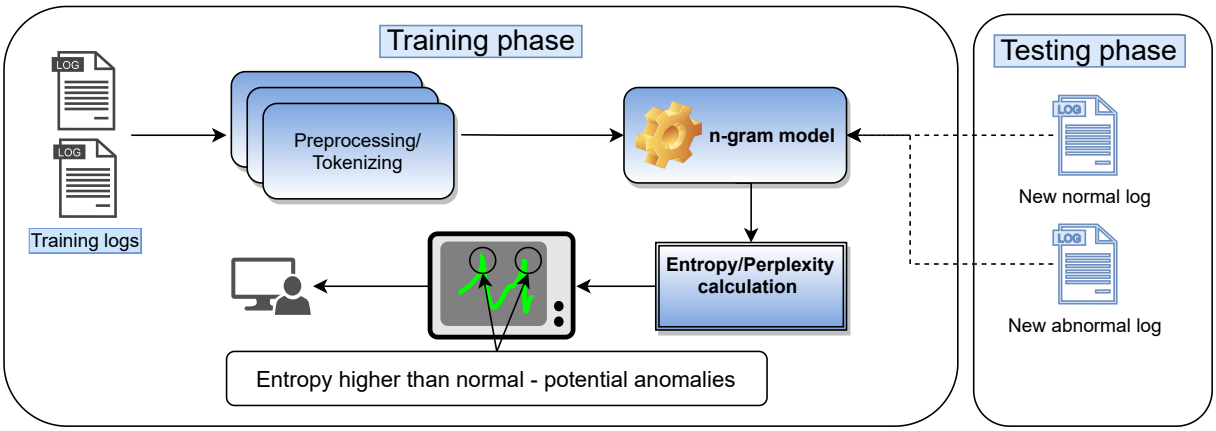



Figure 8.6: ANALOG steps for anomaly detection through log files.

**Usage scenario.** The way we imagine practitioners and developers will use our approach for anomaly detection is as follows: during the ***training phase***, initially, we acquire a set of normal operational log files to train the n-gram model. This process also involves choosing a proper order for the n-gram model (*e.g.*, $n \in (4, 5)$), and detecting the baseline values of entropies. Based on the baseline range of entropy values, we use the *Hampel Filter* approach (Section 8.6.1) to assign a *threshold (Th)* for alarming an anomaly in case the entropy values go beyond *Th* during the testing phase. While in the *testing phase*, as the system is running and continuously generating new log records, we define an *observation window (ow)* for the recently generated logs. The length (in bytes) of *ow* can vary from a single log message up to chuck of log file (*e.g.*, 16 KB), depending on the volume of the logs and the desired granularity of the log analysis. During the continuous testing, the content of *ow* is fed into the NLP model, and the entropies *(E)* are evaluated. If $E > Th$,

this chunk of log data *ow* is isolated as it potentially contains an anomalous log message. Because this usage scenario is an ***online*** approach, the *ow* moves to the next chuck of the log file as soon as new log messages are available. Regarding the granularity of *ow*, ideally, we will be able to test each line of the log file against the NLP model, as it is written to the storage medium, as the testing is quite fast compared to the training of the n-gram model. An incremental enhancement that can be implemented is to periodically retrain the NLP model on the most recent log data once every cycle, such as daily, overnight, weekly, to align the model with recent normal log changes, if any.

Figure 8.7 shows entropy values (*i.e.*, log(*perplexity*)) for distinct orders of n-grams for both normal and abnormal log windows. Solid lines show normal log windows, and dashed lines represent anomalous log windows for 2-, 3-, and 5-gram models. The horizontal axis shows the size of the log window (in KB) that we inspect for perplexity calculation. For creating an abnormal log window, we synthetically inject an error message into the log file. Across the board, once there is an anomaly, the perplexity, and therefore entropy values increase, which signals that we have detected an anomalous log window.

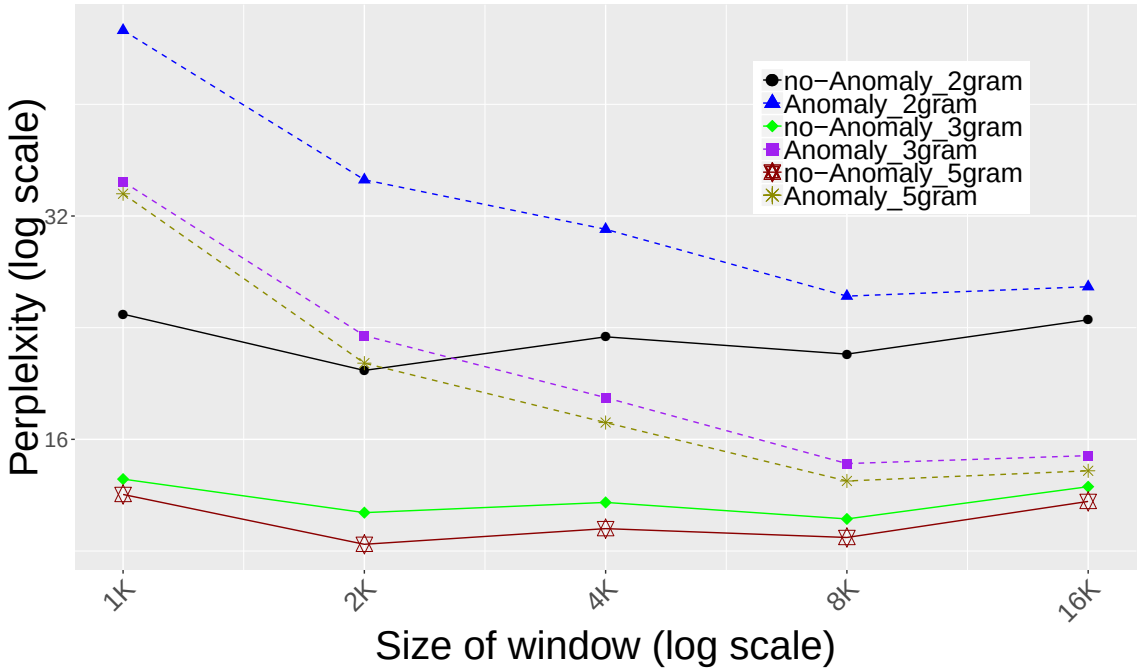



Figure 8.7: Anomaly detection with perplexity values for HDFS.

### 8.6.1 Hampel Filter for Threshold Selection

Hampel filter [249] is a decision filter (*viz., yes or no*) that detects anomalies in the data vector if they lie far enough from the median to be deemed as an outlier. This filter depends on both the entropy vector width and an additional tuning parameter $t$, which explains the margins from the median. For a vector of entropy values ($E = \{e_1, e_2, ..., e_n\}$) measured from the training data, we can calculate the threshold ($Th$) according to the following formula: $Th = median(E) + t \times MAD$, where $MAD$ stands for *median absolute deviation* and is defined as the median of the absolute deviations from the data's median $\tilde{E} = median(E)$: $MAD = median(|e_i - \tilde{E}|)$. The value of $t$ is chosen based on the range of values in $E$, and according to our experiment on the eight projects, it falls in the range of $t \in [1, 8]$. As such, threshold $Th$ is in the range of:

$$\tilde{E} + 1 \times MAD \leq Th \leq \tilde{E} + 8 \times MAD \tag{8.9}$$

Table 8.3 summarizes the $t$ values for different systems. System abbreviations are from Table 8.1. For each project, we evaluated different values of thresholds, and we chose the value of $t$, as an agreeable practice [257], to balance *Precision* and *Recall* and achieve the highest *F-Measure*, which we discuss in the following section. It is admissible to use different values of $t$ to achieve a particular goal, such as maximizing *Precision* or *Recall*.

| System | HS | SP | FW | WD | LS | TB | LB | ST |
|---|---|---|---|---|---|---|---|---|
| ***t*** | 3.0 | 4.5 | 8.0 | 6.0 | 3.1 | 1.7 | 2.8 | 1.4 |

Table 8.3: Values of t for different systems.

### 8.6.2 Evaluation

To evaluate the accuracy of anomaly detection approaches, we use *Precision, Recall, Fall-out, F-measure, and Balanced Accuracy*, which are the commonly used metrics for evaluating anomalies in prior work [337, 151, 346]. These metrics are based on the confusion matrix[3], composed of four values: ❶ true positive ($t_p$) is the correctly identified abnormal log windows by the n-gram model. ❷ False positive ($f_p$) is the number of incorrectly identified normal log windows as abnormal ones. ❸ False negative ($f_n$) is the number of incorrectly identified abnormal log windows as normal ones. ❹ True negative ($t_n$) is the correctly identified not abnormal (*i.e.*, normal) log windows by the n-gram model.

Based on the confusion matrix, the definitions of the metrics are: ***Precision*** is the percentage of log test windows that are correctly identified as anomalies over all the log test windows that are identified as anomalies: $\frac{t_p}{t_p+f_p}$. ***Recall*** (or *true positive rate*) is the percentage of log test windows that are correctly identified as anomalies over all log windows containing anomalies: $\frac{t_p}{t_p+f_n}$. ***Fallout*** (or *false positive rate*) is the percentage of log test windows that are incorrectly identified as anomalies over all normal log windows: $\frac{f_p}{f_p+t_n}$. ***F-Measure*** is the harmonic mean of Precision and Recall: $\frac{2\times precision\times recall}{precision+recall}$. ***Balanced Accuracy (BA)*** is the average of the proportion of anomalous and normal log windows that are correctly classified: $\frac{1}{2}\times(\frac{TP}{TP+FN}+\frac{TN}{TN+FP})$. BA is important for our analysis as only 10% of the synthetically generated cases are anomalous, *e.g.*, $t_n$ is ∼10X greater than $t_p$. In case there is an imbalance in the data (anomalous scenarios happen less often than normal scenarios [151]), BA is widely used [336, 346] for evaluation because it avoids over-optimism that traditional accuracy might suffer from. We perform two sets of experiments: 1) **synthetic anomalies**, 2) **comparison with PCA** [311].

**Synthetic anomalies.** For each log data, we randomly sample 512 MB of logs. We then train a 5-gram model on 90% of the data and keep the rest for testing. Next, we split the test data into 400 sequential samples of 4KB log windows. The idea is to simulate the continuous generation of log windows while the system is running. We then randomly inject anomalous messages in 10% ($anomaly_{freq.} \ll normal_{freq.}$) of the test windows. We calculate the entropy for each log window and detect an anomaly if the entropy is higher than the baseline threshold ($Th$).

**Comparison with PCA.** In this part, we compare our approach with the PCA-based anomaly detection and their provided labeled dataset [311], which is the log of HDFS system operations on various blocks. Although plenty of logs are available, this dataset is one of the few available labeled ones. Initially, to make the log parsing and template extraction manageable (required for [311]), we randomly select 20 MB of log data (∼105K log lines and within 95% confidence [180]) and parse them with IPLoM [217] to extract event log templates belonging to each block (*i.e.*, session; denoted by *blk_-id* in the logs). Then, for each block, we create the sequence of events, *i.e.*, log window, and for each window, a label of normal or abnormal is provided with the dataset. We then split the data in 90%-10% for train-test and evaluate both PCA and ANALOG. PCA, like the majority of anomaly detection methods, requires both normal and abnormal samples to fit log windows into two sub-spaces, *viz.*, normal space, $S_n$, and anomaly space, $S_a$. ANALOG does not require historical abnormal instances, as it builds the n-gram model based on the normal instances, which provides a clear advantage for ANALOG over the majority of the anomaly detection

methods [311, 204, 83, 68], which rely on witnessing anomalous instance in the training data to function.

| System | tp | fp | tn | fn | Precision % | Recall % | Fallout % | F-Measure % | Balanced Accuracy % |
|---|---|---|---|---|---|---|---|---|---|
| HDFS (HS) | 41 | 4 | 356 | 0 | 91.11 | 100 | 1.12 | 95.35 | 94.44 |
| Spark (SP) | 41 | 6 | 354 | 0 | 87.23 | 100 | 1.67 | 93.18 | 99.17 |
| Firewall (FW) | 41 | 3 | 357 | 0 | 93.18 | 100 | 0.83 | 96.47 | 99.58 |
| Windows (WD) | 27 | 6 | 354 | 14 | 81.81 | 65.85 | 1.67 | 72.97 | 82.09 |
| LinuxSyslog (LS) | 35 | 29 | 331 | 6 | 54.69 | 85.37 | 8.05 | 66.70 | 88.66 |
| Thunderbird (TB) | 23 | 24 | 336 | 18 | 48.93 | 56.10 | 6.67 | 52.27 | 74.72 |
| Liberty (LB) | 33 | 15 | 345 | 8 | 68.75 | 80.49 | 4.17 | 74.16 | 88.16 |
| Spirit (ST) | 28 | 22 | 338 | 13 | 56.00 | 68.29 | 1.67 | 61.54 | 81.09 |
| **Average** | 34 | 14 | 346 | 7 | 72.71 | 82.01 | 3.23 | 76.58 | 88.49 |

Table 8.4: Performance of anomaly detection with NLP models for different system logs.

### 8.6.3 Results

**Synthetic anomalies.** Figure 8.8 shows the entropy vector values for the logs of eight systems in our study. The x-axes show the log windows, and the y-axes represent the perplexity ($PP$) values. The horizontal blue line shows the overall trend-line for the $PP$ values of the 400 log windows. The random sudden spikes in the $PP$ values indicate that the n-gram model considers the content of these log windows surprising and not similar to the normal logs. As such, it identifies them as abnormal. Table 8.4 shows the result of our anomaly detection for different systems. Figure 8.9 plots *receiver operating characteristic* (ROC) graph, which is the *true positive rate* against *false positive rate*. ROC illustrates the ability of our approach in classifying anomalies. The black line corresponds to randomly classifying normal and abnormal log windows, and the red dot on the top left corner shows the perfect classifier (PC), and the blue labels show the performance of ANALOG for eight evaluated projects (two labels, $SP$ and $ST$, are overlapping). Our evaluations all fall on top of the random line, and closer to PC.

**Comparison with PCA.** In the second part of the experimentation, we compare our approach with the PCA-based anomaly detection approach proposed in [311]. Fig.8.10a shows the perplexity values for 795 test samples of log windows, with 32 anomalies gathered on the left side of the chart, and 763 normal samples. As observed from the blue trend line, the perplexity is lower for normal samples. Fig.8.10b summarizes the evaluation metrics for ANALOG, which are higher when compared with PCA across the board. We rationalize that because ANALOG assigns probabilities for the sequences of log events, *i.e.*, the sequences of n-grams, it can better distinguish normal and abnormal sequences when

compared to PCA. We believe our anomaly detection results in Table 8.4 and Fig. 8.10 are encouraging, and we expect further adaptation and continuation of our NLP-based research for other software engineering tasks.

**Summary of the findings.** Lastly, Table 8.5 provides a synopsis of our main findings and their implications from studying the NLP characteristics of logs.

## 8.7 Related Work

**NLP in software engineering tasks.** Prior work has considered natural language processing techniques for software engineering (SE) tasks. Initial SE research in this field [119, 155] showed that the software's source code is regular and repetitive. As such, these works aimed to represent the sequences of the source code with n-gram language models. Consequently, applications of this representation emerged in software-related tasks such as bug reports [137], identifier name, class name, and next token code suggestion [48, 49, 66]. Following the work of Hindle *et al.* [155], Tu *et al.* [298] explored the idea of the localness of the source code and leveraged it to improve the performance of n-gram language models by introducing a local cache. Focusing on the source code's logging statements, He et. [145] investigated the NLP features of logging descriptions and used them to answer the question of *"what to log?"*. Inspired by and orthogonal to prior work, in this research, we utilize n-gram models to characterize the naturalness and localness of log files and to improve automated log analysis by leveraging these attributes.

**Automated log analysis.** As software logs contain rich information on the runtime state of the systems, their analysis has been the focal point of various prior research to improve systems' reliability and user experience, such as anomaly detection [311, 116], user statistics [187], fault detection and diagnosis [349]. These works typically employ data mining and learning algorithms to efficiently analyze a large scale of logs, which also involves log collection [104] and log parsing [146]. For example, Xu *et al.* [311] leveraged a dimension reduction algorithm (PCA) to distinguish normal and abnormal events in distributed systems. In this study, however, we propose to uncover the NLP characteristics of logs, and we leverage them for benefiting automatic log analysis.

**Anomaly detection with logs.** Anomaly detection refers to the task of uncovering events that do not follow the expected behavior in the system [76]. Possibly, an anomaly in the system might turn to a fault, an error, and eventually to a system failure, if left untreated [56]. As such, log files, because of their rich content of runtime information and events, are widely utilized for computer systems' anomaly detection [68, 337, 69, 107]. Different from the prior work, we present an NLP-based method to investigate anomalies in

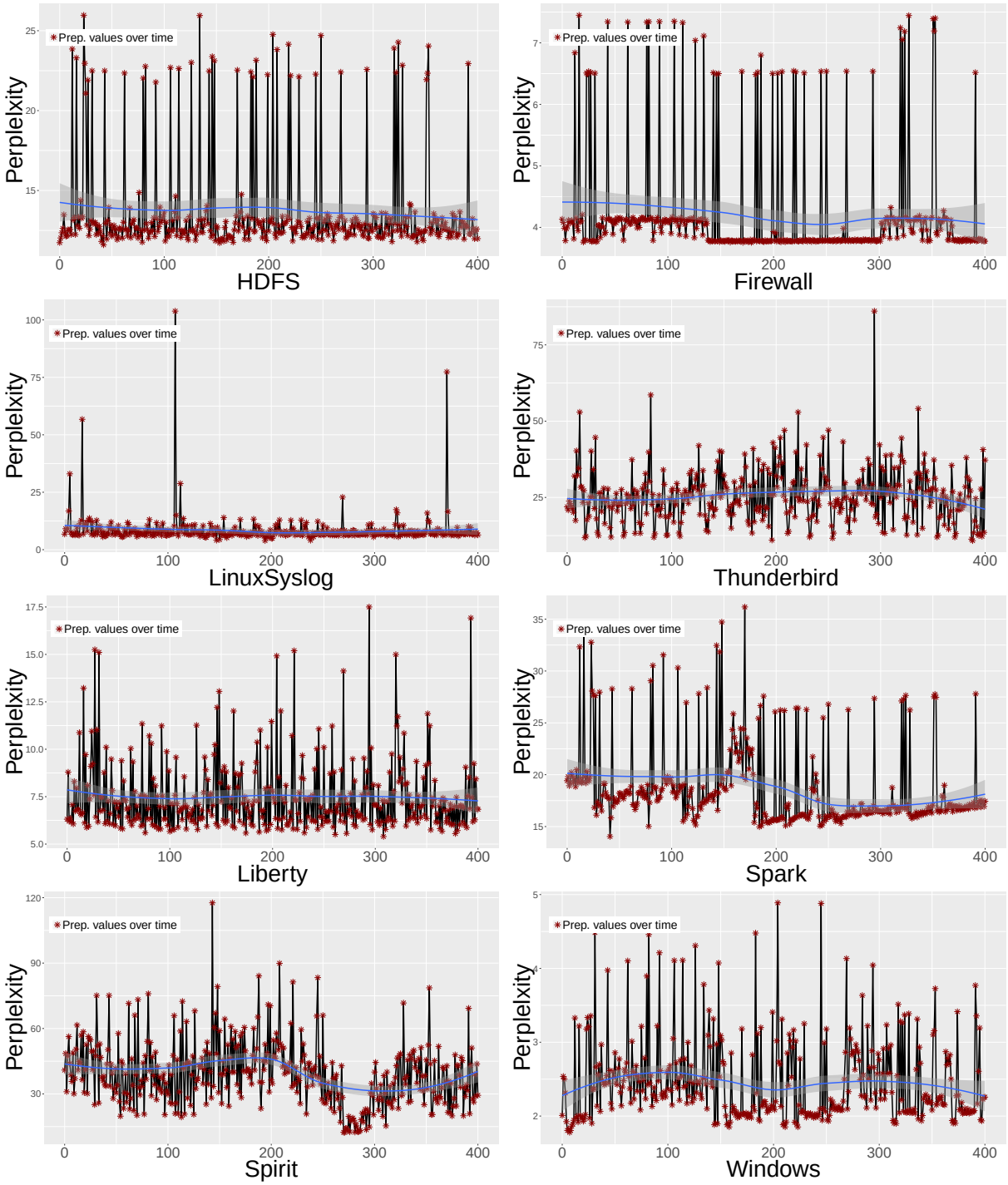



Figure 8.8: Perplexity values for different projects.

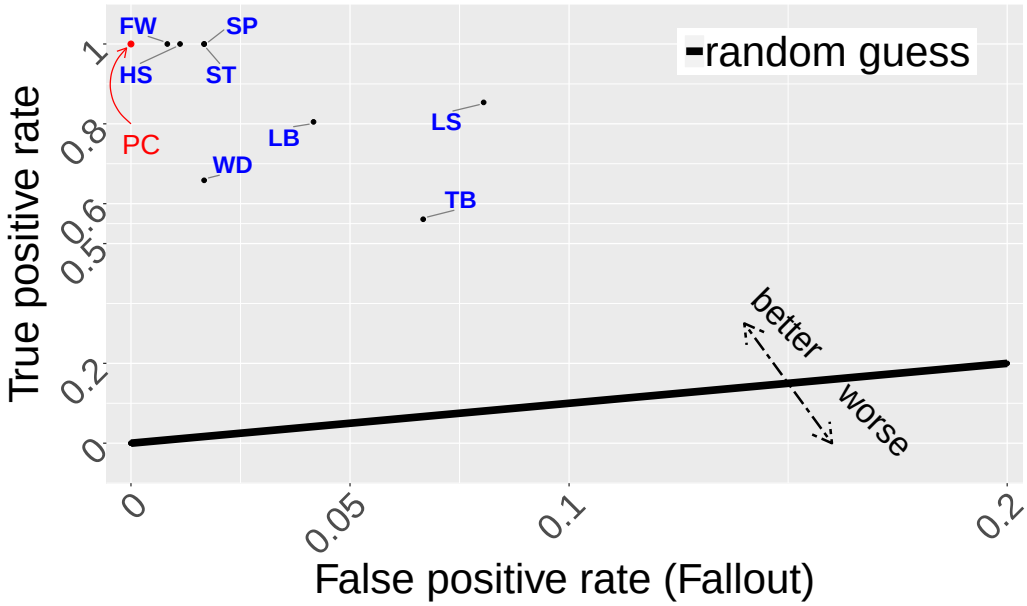
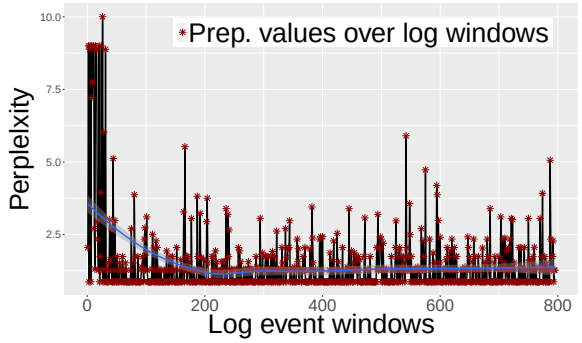
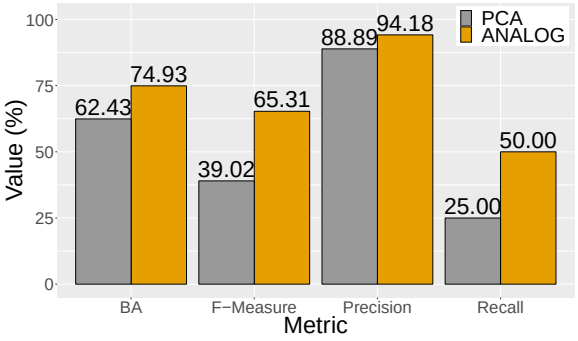



Figure 8.9: ROC for performance of ANALOG. The black line (random guess) shows $y = x$, *i.e.*, 45°angle; however, the x-axis is stretched to provide a better separation on the ROC values of the evaluated projects.




(a) Perplexity values for log windows. Anomalous logs are presented at the beginning of the graph to show how overall perplexity trend changes for abnormal against normal logs.




(b) This chart compares the values for *Balanced Accuracy (BA)*, *F-Measure*, *Recall*, and *Precision* for ANALOG vs. PCA.

Figure 8.10: PCA versus ANALOG performance comparison.

| Research questions (RQs) | Findings (Fs) | Implications (Is) |
|---|---|---|
| **RQ1:** does a natural repetitiveness and regularity exist in log files? | **F1:** language n-gram models capture a high degree of repetitiveness in software systems' logs, even to a higher degree than in common English corpora. | **I1:** compared with common English, the repetitiveness of log data can be better captured by statistical language models. |
| **RQ2:** is the regularity that the statistical language models capture merely because of the limited language of logs, or it is also a project-specific characteristic? | **F2:** n-gram-based statistical language models capture a high level of regularity and repetitiveness in within-project analysis and less cross-project regularity. | **I2:** the lower entropy values of log data are the outcome of the predictable repetitiveness within each system's logs, and not the limited language of logs. |
| **RQ3:** how does Zipf's law capture the repetitiveness of high-rank tokens in log files? | **F3:** from Zipf's law calculation, log files illustrate a higher concentration of high-rank tokens when compared to the English corpora. | **I3:** logs are more predictable, and language models potentially perform better in predicting the next token of a sequence when applied on log data. |
| **RQ4:** are log n-grams *endemic* to their projects? | **F4:** a significant percentage of endemic n-grams are repetitively used in the local context of logs. | **I4:** log files are locally endemic. Endemic n-grams in logs are the artifact of endemic n-grams in the source code and software itself. |
| **RQ5:** are log n-grams *specific* to their projects? | **F5:** each system tends to use its own set of n-grams more frequently, which is reflected in its logs. | **I5:** log files are locally specific. **F4** and **F5** enable more efficient analysis of logs with localized caching models. |
| **RQ6:** how the logs' naturalness and localness can help with the automated analysis of the log files? | **F6:** with an n-gram model that is trained on normal logs, abnormal logs result in higher-than-normal entropy values during testing. | **I6:** this finding opens up an avenue of research to utilize NLP attributes for automated log analysis tasks, such as anomaly detection. |

Table 8.5: Summary of RQs and our findings.

software logs. As an advantage, because we utilize NLP features of logs, our approach does not require a log parsing step, which is the first step of every log analysis approach [146]. Two main categories of anomaly detection include supervised (*e.g.*, [68, 116, 107]) and unsupervised (*e.g.*, [311, 207, 212]) methods. In industry settings, a system may encounter only very few anomalies per year [151]. As such, a long list of prior methods [68, 311, 204, 334, 65], which relies on seeing anomalous instances in the training data, becomes ineffective. We believe because ANALOG does not require historical abnormal instances, as it builds the n-gram model based on the normal instances, it has an edge on such approaches.

## 8.8 Threats to Validity and Discussion

***External threats*** to the validity reflect on the generalization of our work to other such software projects and log files. In this research, we conducted our NLP analysis on logs of eight software systems. We picked the systems from different domains and assumed our approach is independent of the underlying programming language, source code, and the software architecture that the system is implemented with. However, since other software systems may follow different logging practices, our findings may not accurately extend and generalize to logs of such other systems. Regarding ***internal threats*** to the validity, we rely on the accuracy of the n-gram model to calculate the entropies. Additionally, the threshold selection, and false positive and false negative values can affect the accuracy of our anomaly detection approach. Finally, due to the limited availability of labeled datasets, although we conducted experimentation on HDFS and synthetic anomaly datasets, future opportunities that give access and enable us to evaluate ANALOG on logs of large-scale and enterprise software systems would further solidify our findings.

## 8.9 Closing Remarks

This chapter explores the natural language attributes of software logs. Guided by a set of research questions, our findings confirm that log files, as an artifact of software systems, are natural and local, even more so than common English corpora. We show how the NLP characteristics of log files can be leveraged for log analysis tasks, and we present *ANALOG*, an anomaly detection tool that is built upon NLP features and outperforms the prior work.

In this research, our primary focus has been to show the NLP characteristics of the logs and illustrate the potential applications of NLP for log analysis while not limiting our approach to anomaly detection techniques. Therefore, as a future direction, besides

anomaly detection, we look into extending our work to other software analysis tasks, such as extracting system security infringements from the system and network logs. Additionally, we also aim to leverage deep learning (DL) language models to potentially improve our approach and have a comparison against DL anomaly detection approaches.

# Chapter 9

# L'PERT: Log Parsing with BERT

**Abstract-** As software systems continuously record important runtime information for various dependability analyses in unstructured log records format, log parsing has become a crucial first step to extract organization and interpret the log lines in an automated fashion. As log parsing is the prerequisite of downstream log analysis tasks, *e.g.*, anomaly detection, even small inaccuracies in log parsing can hinder and ultimately nullify the benefits of downstream automated log analysis tasks. Thus, the design and performance improvement of log parsing tools remains a consistent challenge. As log lines show natural language (NLP) attributes, the recent advancement of natural language models motivates the application of them for achieving more accurate log parsing. Thus, in this research, we investigate the application of the state-of-the-art language models, *i.e.*, transformers, for log parsing. We also show how our approach provides additional flexibility in the tokenization step and its design when compared to prior work. In sum, our research enables the application of transformer-based language models for log parsing and exhibits promising performance.

**Keywords:**
log parsing, software systems, transformers, BERT, language models, natural language processing

## 9.1 Introduction

As computing systems continue to grow not only in size but also in the number of their users, their management, maintenance, and dependability analysis become more daunting

and calls for more sophisticated approaches. Therefore, system operators and practitioners more recently look for AI-enabled solutions to facilitate more effective and scalable analysis of large-scale computing systems. A common way to enable system dependability analysis is through the runtime software execution logs [347]. The rich information that resides in the logs plays a crucial role in runtime or postmortem dependability analysis of computing systems, and has enabled various system dependability and management tasks such as anomaly detection [107, 337, 224, 133, 160], user statistics [187], application security [244], performance problems and system failures [104, 321, 341].

Although logs contain valuable information, efficient and effective analysis of logs remains a twofold challenge:

❶ Due to the increasing size of software systems and their user base, they tend to generate a significant amount of logs. Research [347] has reported that business cloud environments generate several terabytes of log data every single day. The sheer size of logs makes it practically impossible to rely on traditional and manual ways of looking in logs for problem diagnosis.

❷ As manual analysis of logs becomes ineffective, proposing automated log analysis tools is still a daunting challenge. Log records are intrinsically un- or semi-structured statements inserted by developers in the source code. Meaning that logs are primarily designed to be a natural language sequence [145, 126], conveniently readable and fathomable by a human log reviewer. This adds additional difficulty and puts it at odds with automated analysis of logs.

Motivated by the aforementioned challenges, many prior efforts both in academia, *e.g.*, Deeplog [107], and industry, *e.g.*, ELK Stack[30] and Splunk [43] have examined approaches and proposed tools to automate log analysis. Prompted by the need for effective and efficient downstream log mining tasks, *"log parsing"* is the first major step to enable automated log analysis. Log parsing is essentially the task of extracting organization (*i.e.*, *log template*) from raw log records to make them machine readable, *i.e.*, in contrast to the free-form human-readable log records. A log parser aims to extract the logging template for the corresponding line of the source code that the logging statement is originated from.

Traditionally, log parsing methods relied on handcrafting *regular expressions (RegEx)* by a system expert with domain knowledge. This way of hard coding RegExes can no longer be leveraged as different modules of large-scale software systems often generate logs in different formats, and it is also hardly possible to find an expert knowledgeable and responsible with the entire system. Additionally, log statements tend to continuously evolve [277]. As such, hard coding the templates renders them inflexible to these

changes. Several log parsing approaches apply various clustering approaches to extract templates [116, 294, 230, 284, 139]. Other approaches utilize associate rule and frequent pattern mining [232, 236], an evolutionary approach [225], and longest common subsequent approach [106]. Heuristic-based approaches, such as Drain [148], produce the best results when compared to other approaches [347]. More recently, deep-learning neural-based approaches [238] have shown superior performance in the log parsing task.

As log messages are natural language [145, 126], and in parallel, because of recent advancements of NLP models [301, 101], we hypothesis that the application of NLP approaches has the potential to result in more accurate log parsing. As such, in this research, we employ transformer models for extracting log templates. We leverage **M**asked **L**anguage **M**odel (MLM) task to extract log templates. This task aims to predict the tokens that belong to the log template by masking them and then predicting them with a high probability. We also discuss how our approach brings additional flexibility by removing the required domain knowledge in defining RegExes and its design.

The remainder of the chapter is organized as follows. Section 9.2 provides the necessary background, and Section 9.3 explains our approach for log parsing. In Section 9.4, we present an evaluation of our study compared to prior work, and we discuss some of the advantages of our work in Section 9.5. We review the prior work in Section 9.6, and finally, we conclude this chapter in Section 9.7.

### 9.1.1 Contributions

The main contributions of our research in this chapter are:

1. We propose L'PERT, a novel deep learning-based approach that can extract log templates without any supervision and system knowledge.
2. We then evaluate the performance of L'PERT with prior work and discuss the flexibility that our approach brings in design and tokenization.

## 9.2 Background

The goal of log parsing is to extract log templates. Put formally, a temporally ordered log sequence, $ls_k$, of log messages, $m_i$s, for a specific run, $k$, of the system is a subset of the entire finite set of log messages ($LS$) available for the system. Each $m_i$ corresponds

to a specific log statement in the source code, *e.g., log.info()* and *printf()*. $m_i$ consists of a sequence of ordered tokens, $t_i$s, some of them belonging to the static content of log statement, $t^s_{ij}$ of log statement, and some of them are dynamic variables, $t^d_{ij}$. $j$ is the index of the token in $t_i$, *i.e.*, $j = 0, 1, 2, ..., N$ and $N$ is the length of $t_i$, $N = |t_i|$. Thus the task of log parsing is to for each log token sequence, $t_i$, find the tuple, $(t^s_i, t^d_i)$, such that:

$$\forall m_i \in ls_k : parse(t_i) - - > (t^s_{ij}, t^d_{ij}), j = 0, 1, 2, ..., N$$

For example, for the logging statement from Blue Gene/L (BGL) supercomputer logs [241, 151] *"CE sym 20, at 0x1438f9e0, mask 0x40"*, the template is *"CE sym ⟨∗⟩, at ⟨∗⟩, mask ⟨∗⟩"*. Static parts of the log message are reflected without change and the dynamic parameters of the logging statement, *i.e.*, *[20, 0x1438f9e0, 0x40]* are substituted with ⟨∗⟩, as the same logging statement in the source code can be printed with different dynamic parameter values in its every iteration. The better the log parser can distinguish the static parts and dynamic values of the log message, the higher the quality of log parsing.

## 9.3 Approach

Figure 9.1 shows the high-level design of L'PERT. The input log line is *"162 duble-hummer" alignment exceptions"*.

### 9.3.1 Pre-processing

In order to prepare the log data to be fed into the learning process, first we extract the content of each log message from the log header. In the provide example above, the log line from the log file is: *- 1117842974 2005.06.03 R24-M0-N1-C:J13-U11 2005-06-03-16.56.14.254137 R24-M0-N1-C:J13-U11 RAS KERNEL INFO 162 double-hummer alignment exceptions*. The log header generally contains the source id, date and timestamp, and the log verbosity level. The separation of the header from content is consistent with the common practice in the prior work [347, 238]. In addition, the header log template is the same for all log messages, *i.e.*, it is trivial, and it is decided by the logging library, more specifically, the logging library appenders [37].

### 9.3.2 Tokenization

After the log content is extracted for each log line, we tokenize the log content to its individual tokens by whitespaces. All the tokens are added to build a vocabulary for each

system's log. The start and the end of the log message are marked with two dedicated tokens, [SOS] for the start of the string, and, [EOS], for the end of the string. The idea is to enable the model to clearly distinguish the start and the end of each log message. During the training, each token of the log message is being masked ([MASK]) for each iteration. The goal is to train the transformer model to predict the masked token by paying attention to the other tokens that exist in the context of the log message.

We also augment our tokenization process with Byte-Pair Encoding (BPE) [276] to enable a suitable balance of intra-word tokenization. This also makes our approach more flexible than prior work [238] that uses a set of pre-defined filters (*i.e., RegExes*) to tokenize log statements. The filters are system-dependent and require some domain knowledge. Thus, our approach has an obvious advantage over prior work and it can parse a new system's log without requiring hard-coding log message filters.

### 9.3.3 Word Embedding and Axial Positional Embedding

After tokenization, we embed the tokens to vector of values. Because all of the operations in transformers happen on numerical values, we initially convert tokens to embedding vectors. Each token in the input vocabulary (*e.g.*, in our case, the vocabulary of the entire log sequences) is represented by a unique embedding vector. The embedded vectors are numerical arrays $X \in \mathbb{R}^d$, *i.e.*, $X = (x_1, x_2, ..., x_d)$ where $d$ is the size of the embedding layer, *e.g.*, $d == 256$ in our design. The values of the embedded vectors are adjusted during ***training phase*** to minimize the model loss function. Following the word embedding layer, we apply axial embedding to encode the positional embedding of tokens.

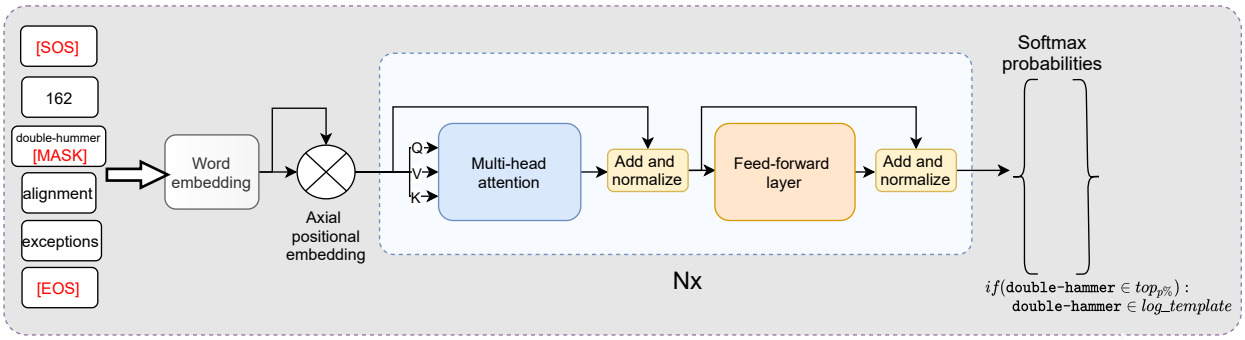



Figure 9.1: High level design for L'PERT.

Because the attention-based model does not have a mechanism to incorporate the order

of the tokens in the sequence and performs the computation for all the tokens in parallel, we leverage axial positional embedding (APE) [157, 176] to preserve and incorporate the positional order of tokens in the log messages. As such, APE encodes and integrates the positional information with word embedding to preserve the context. Assuming $X$ is the word embedding vector, and $X'$ will be the output after the APE module, then: $X' = APE(X) + X$

### 9.3.4 Multi-Head Attention and Feed Forward

The output of the APE is fed into the multi-head attention layer. Intuitively, the multi-head attention finds the relationship of the embedded vector for the masked token with the embedded vectors for other tokens in the log message. The multi-head aspect of the design allows the model to pay attention to multiple tokens simultaneously, instead of paying attention to the token relationships individually. The attention score is evaluated by multiplying the query (Q), value (V), and key (K) matrices as the inputs of the attention layer. The attention layer is followed by ***add and normalize*** (AddNorm) layer. This layer is leveraged to improve the performance of the transformer to address training phase deficiencies such as vanishing gradients [93] and covariate shift [163]. Assuming the output of the attention layer is $X''$, thus, the output of the add and normalize layer is $X''' = norm(X' + X'')$.

Following the AddNorm layer, there exists a position-wise feed-forward (PFF) layer, which comprises a fully-connected feed-forward neural network that enables the model to calculate the combination of different attentions. Finally, the weights of the model are passed through to another AddNorm layer followed by a Softmax layer. In the Softmax layer, we look for the top $p\%$ of the tokens which are predicted as the [MASK] token. If the masked token (*e.g.*, double-hummer) is among them, we then classify the masked token as a static part of the log template. Otherwise, the masked token is considered a dynamic parameter of the log message and is replaced with $\langle * \rangle$ in the log template. The optimal value of $p$ is selected during hyperparameter tuning as we train the model with different values of $p$ and evaluate the results. Our approach provides additional flexibility in this regard compared to prior work that introduces project-dependent thresholds [238].

The block that contains multi-head attention, PFF, and AddNorm layers (highlighted with light blue) can be repeated $N$ times ($Nx$), to achieve a deeper model and potentially improve the performance of the learning task, which simultaneously lengthens the training time. In our design, $N == 1$, and deeper models are left as future work.

## 9.4 Evaluation

We guide our evaluation with the following research question (RQ): *How effective is L'PERT in log template extraction?*

### 9.4.1 Evaluation Dataset

We evaluate our approach on the BGL dataset that is collected by Lawrence Livermore National Labs (LLNL) from the BlueGene/L supercomputer system.

### 9.4.2 Evaluation Metrics

To measure the effectiveness of L'PERT, we calculate the Precision, Recall, F-Measure, Parsing Accuracy, and Edit Distance scores for log parsing. These scores are defined as follows:

- **Precision**. The ratio of correctly detected log templates amongst all detected log templates is $Precision = \frac{TP}{TP+FP}$.
- **Recall**. The ratio of correctly detected log templates over all ground truth log templates is $Recall = \frac{TP}{TP+FN}$.
- **F-Measure**. The harmonic mean of Precision and Recall is $F-Measure = \frac{2\times Precision\times Recall}{Precision+Recall}$.
- **Parsing Accuracy (PA)**. The ratio of correctly parsed log messages over the total number of log messages. We consider a log message correctly parsed if and only if its template belongs to the same group of logs as the ground truth [347]. For example, if a ground truth log sequence [E1, E2, E1] is parsed as [E1, E2, E3], then $PA$ is $\frac{2}{3}$, since the 1st and 3rd log messages are not grouped together. The institution is that we evaluate the ratio of logs that belong to the same template (*i.e.*, cluster) in ground truth are also detected to belong to the same template in the parsed logs.
- **Edit Distance (ED)**. Although PA is commonly used for evaluating the performance of clusters, it does not capture the similarity of the parsed log templates to the ground truth log templates. Thus, it is inaccurate as it does not capture the differences that might exist in the tokens of each log template. For example, if the log parser parses all the occurrences of "CE sym ⟨∗⟩, at ⟨∗⟩, mask ⟨∗⟩" as "⟨∗⟩ ⟨∗⟩ ⟨∗⟩,

⟨∗⟩ ⟨∗⟩, ⟨∗⟩ ⟨∗⟩", PA achieves perfect accuracy, despite the obvious error. Thus, in our design we focus on minimizing the Levenshtein edit distance [190], *i.e.*, the exact number of character changes that are required to convert the parsed log template to the ground truth template. As such, the lower the ED, the higher the similarity of the parsed template and the ground truth.

Table 9.1 summarizes the result for BGL dataset for both L'PERT and Nulog.

## 9.5 Discussion

### 9.5.1 Flexibility

An important point to emphasize is that our approach does not require any domain knowledge about the format of log statements and does not receive any RegExes on possible log patterns of log messages, *i.e.*, fully unsupervised. On the contrary, prior work relies on regular expressions and filters on log messages format for each system log to perform tokenization. As such, we observe the significant flexibility that our approach brings, and, simultaneously, achieves comparable log parsing performance when compared to prior work.

In addition, regarding the threshold selection for [MASK] token identification, our approach provides additional flexibility compared to prior work that introduces project-dependent thresholds [238].

### 9.5.2 Tokenization

Through manual scrutiny of some of the inaccurately extracted log templates, we observe that some of the inaccuracies are because of wrong tokenization. One key observation is that punctuation marks generally have different interpretations in log files compared to general text. For example, in IP addresses that often happen in log messages (*e.g.*, 10.251.91.229), the ground truth for the template is ⟨∗⟩. However, because the general-purpose tokenizer treats '.' as a punctuation mark, the extracted template is ⟨∗⟩.⟨∗⟩.⟨∗⟩.⟨∗⟩. For example, almost all HDFS log lines contain an IP address, as data is transferred from a source IP to a destination IP. As such, for these systems, the exact match accuracy is low, although the log parser has done a satisfactory performance in detecting the overall log template. This also brings attention to the need for the development

| Parser | Precision | Recall | F-Measure | PA | ED (mean) |
|---|---|---|---|---|---|
| Nulog | 1.0000 | 0.9992 | 0.9996 | 0.9550 | **2.4365** |
| L'PERT | 1.0000 | 0.9969 | 0.9985 | 0.9425 | **3.096** |

Table 9.1: Log parsing results for BGL dataset.

of a log-aware tokenizer that understands and treats punctuation marks in the context of logs and not in general-purpose English text.

## 9.6 Related Work

### 9.6.1 Log Parsing

Prior research has dedicated a noticeable number of studies to propose different techniques for log parsing [347, 146]. He *et al.* showed the importance of log parsing, and that even small errors as low as 4% in the performance of log parser in template extraction can result in an order of magnitude degradation in the performance of downstream log mining tasks. As such, many of the prior work aims to improve the log template extraction accuracy with different approaches.

To overcome the shortcomings of prior work, we propose L'PERT that is different from prior work as it uses a self-attention mechanism from transformers. L'PERT leverages masked language model (MLM) task from transformers architecture by masking every single token in the log message and then it looks for predicting it based on its *"attention"* with other tokens in the log message. Our evaluation illustrates that L'PERT achieves comparable performance with prior work and brings in additional flexibility.

### 9.6.2 NLP for Software Engineering Tasks

Prior research has shown that logging statements within the software's source code and their end-products, *i.e.*, logging messages in the log file exhibit natural language characteristics [126, 145]. More specifically, logs are natural and local, and are even more repetitive than natural text. This implies that the sequence of log tokens in log messages within log files can be predicted based on the context surrounding the tokens. As such, we utilized the state-of-the-art NLP models to perform log parsing. Recent advances in NLP models [301, 101] have shown that attention-based transformers have outperformed

Long Short-Term Memory (LSTM) [158] and other traditional NLP models in a variety of NLP tasks. Transformer-based NLP models consider the content from both left and right of the token and can pay attention to multiple points in the log message simultaneously, whereas LSTM-based models only consider the preceding context while making a decision for the token under consideration. As such, transformers incorporate more context and semantic from log sequences, and thus, outperform LSTM designs in NLP tasks. Recently, Nedelkoski *et al.* [238] have introduced the application of attention-based models for log parsing with *Nulog*. In contrast, our approach performs fully unsupervised log parsing, whereas prior work relies on domain-dependent regular expressions to tokenize the log messages. Thus, our approach brings a higher degree of flexibility and removes the manual process of RegEx definition, and still achieves comparable performance.

## 9.7 Conclusion and Future Directions

In this chapter, we showed how transformer-based log parsing can incorporate additional log message context and perform better in log parsing. We introduced L'PERT, a log parsing tool that applies attention-based transformers to achieve high parsing performance and removes the need for manual filters for log tokenization. As future work, we look into fine-tuning transformers by using pre-trained BERT models.

# Part V
# Epilogue

# Chapter 10

# Conclusions and Future Work

In this thesis, we studied approaches for automating logging statements and analysis of logs with data mining and machine learning approaches. For this purpose, we started with explaining the importance of logging, log statements, and log files, and then proposed novel approaches for automating the logging processes. Our research provides significant value as it focuses on software logs, which play an important role in system debugging and dependability analysis.

## 10.1 Summary of Findings

### 10.1.1 Part II

We initially conducted a comprehensive systematic literature review and mapping of log-related research in Chapter 2. Our findings in this chapter shed light on the ongoing logging research and provide promising opportunities for future work.

### 10.1.2 Part III

As we uncovered some of the challenges of logging, in Part III of the thesis and in Chapters 3, 4, 5, and 6, we tackled the log statement automation challenges with the help of similar code snippets. More specifically, we provided deep learning and NLP-based techniques to predict the location and content of logging statements.

### 10.1.3 Part IV

In Part IV of the thesis, we provided an analysis of cost and gain associated with logging in distributed systems with and without failures in Chapter 7. Our findings provide insight for developers and practitioners on selecting the proper log verbosity level and then how to connect that with the amount of information gained for each log verbosity level.

### 10.1.4 Part V

Next, in Part V of our research, we showed the natural language characteristics of logs and how they can be leveraged for automated analysis of logs in downstream log mining tasks in Chapter 8. We then showed how NLP characteristics and language models can be applied for log parsing in Chapter 9. More specifically, we applied the SOTA transformers architecture for this task.

## 10.2 Avenues for Future Work

### 10.2.1 Chapter 2 - Survey of Logging Research

Our survey on logging research in software systems uncovered several avenues for future work. For example, future research can consider and tackle challenges in *adaptive logging*, *logging practices*, *automated and constraint-based log generation* and *automated analysis of log files*. The continuation of these research avenues will help to improve the quality of log statements and log files, and thus will enable more effective log analysis tasks.

### 10.2.2 Chapters 3, 4, 5, and 6 - Log Statement Prediction

Our work on automating log statements can be further augmented with predicting other details of log statements, such as log verbosity level and its variables. In addition, a potential way to improve our proposed approach is to not only train the NLP model on the log descriptions, but also on the source code context surrounding the logging statements. For example, prior research has used the source code context to suggest auto-generated comments [159].

### 10.2.3 Chapter 7 - Logging Cost and Benefit

The work on logging cost and benefits can be extended to compare quantitative measures of logging cost with qualitative measurements, *i.e.*, surveying developers' opinions on log-related costs and benefits. In addition, future work can look into evaluating logs of other software systems, and investigate how the information gain metric can be translated to more effective troubleshooting by leveraging the combination of execution logs and system runtime metrics.

### 10.2.4 Chapter 8 - Naturalness and Localness of Logs

The work on NLP models for log analysis can be extended to capture semantics from log files and apply that for sophisticated log analysis such as anomaly or failure detection. In addition, as a future direction, besides anomaly detection, research efforts can look into extending our work to other software analysis tasks, such as extracting system security infringements from the system and network logs. Moreover, future work may also consider leveraging deep learning (DL) language models, in lieu of the n-gram model, to potentially improve our approach and have a comparison against other DL anomaly detection approaches from logs.

### 10.2.5 Chapter 9 - L'PERT

Future research can investigate further fine-tuning of transformers by using pre-trained BERT models. Potentially, this will allow for more accurate log parsing. In addition, domain-specific pre-training of BERT models, *i.e.*, training on a large set of software logs, can further enhance their effectiveness for log parsing tasks. This is because we showed that there are some differences between log messages, and normal text, *e.g.*, '.' punctuation mark in $\langle * \rangle.\langle * \rangle.\langle * \rangle.\langle * \rangle$ for IP addresses has a different meaning than normal text. Thus, we hypothesize starting from a large database of system logs can be beneficial for pre-training of BERT models as an avenue for future work.

# References


[1] Abstract syntax tree. https://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/.

[2] Amazon's one hour of downtime on Prime Day may have cost it up to $100 million in lost sales. https://www.businessinsider.com/amazon-prime-day-website-issues-cost-it-millions-in-lost-sales-2018-7.

[3] Confusion matrix. https://en.wikipedia.org/wiki/Confusion_matrix.

[4] cross-entropy. http://en.wikipedia.org/wiki/Cross_entropy.

[5] Data for log naturalness analysis. https://github.com/sgholamian/naturalness_of_software_logs.

[6] Dataset for logging cost research. https://github.com/sgholamian/logging_cost.

[7] Markov property. https://en.wikipedia.org/wiki/Markov_property.

[8] Mira System. https://www.alcf.anl.gov/alcf-resources/mira.

[9] N-gram model. https://en.wikipedia.org/wiki/N-gram.

[10] Project Gutenberg. https://www.gutenberg.org/.

[11] Project Gutenberg on Wikipedia. https://en.wikipedia.org/wiki/Project_Gutenberg.

[12] Replication package for log-aware clone detection. https://github.com/sgholamian/log-aware-clone-detection.

[13] Spark scheduling. https://spark.apache.org/docs/latest/configuration.html#scheduling.

[14] When It Goes Down, Facebook Loses $24,420 Per Minute. https://www.theatlantic.com/technology/archive/2014/10/facebook-is-losing-24420-per-minute/382054/.

[15] The Apache Software Foundation. logging services project. http://logging.apache.org/, 2019.

[16] Apache Spark. https://spark.apache.org/, 2019.

[17] Monitoring Petabytes of Logs per Day at eBay with Beats. https://www.elastic.co/blog/monitoring-petabytes-of-logs-at-ebay-with-beats, 2019.

[18] The AspectJ Project. https://eclipse.org/aspectj/, 2019.

[19] TPC Benchmark. http://www.tpc.org/tpcw/, 2019.

[20] Apache Software Foundation. www.apache.org, 2020.

[21] Centralize, transform and stash your data. https://www.elastic.co/products/logstash, 2020.

[22] Elasticsearch, the heart of the Elastic Stack. https://www.elastic.co/elasticsearch/, 2020.

[23] Hadoop Distributed File System. hhttps://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, 2020.

[24] Jira — Issue and Project Tracking Software. https://www.atlassian.com/software/jira, 2020.

[25] Logback Logging Framework. http://logback.qos.ch/, 2020.

[26] Simple Logging Facade for Java (SLF4j). http://www.slf4j.org/, 2020.

[27] The Apache Hadoop Project. https://hadoop.apache.org/, 2020.

[28] The Epic Guide to Artificial Intelligence for DevOps Automation. https://www.targetprocess.com/blog/artificial-intelligence-devops-automation/, 2020.

[29] The Simple Logging Facade for Java (SLF4J). www.slf4j.org, 2020.

[30] What is the ELK Stack? https://www.elastic.co/what-is/elk-stack, 2020.

[31] Your window into the Elastic Stack. https://www.elastic.co/kibana, 2020.

[32] Datadog: Cloud Monitoring as a Service. https://www.datadoghq.com/, 2021.

[33] Datasets. https://github.com/logpai/LoggingDescriptions, 2021.

[34] Dynatrace: The Leader in Cloud Monitoring. https://www.dynatrace.com/, 2021.

[35] Fast, powerful searching over massive volumes of log data. https://www.loggly.com/, 2021.

[36] Keras: the Python deep learning API. https://keras.io/, 2021.

[37] Log4j Appenders. https://logging.apache.org/log4j/2.x/manual/appenders.html, 2021.

[38] LogPai Log Parse Benchmark. https://github.com/logpai/logparser, 2021.

[39] NewRelic: Real-time insights for the modern enterprise. https://newrelic.com/, 2021.

[40] OpenStack Fault Injection Dataset. https://github.com/dessertlab/Fault-Injection-Dataset, 2021.

[41] TeraSort for Spark. https://github.com/ehiggs/spark-terasort, 2021.

[42] TestDFSIO for Spark. https://github.com/BBVA/spark-benchmarks, 2021.

[43] The Data-to-Everything Platform Built for the Cloud. https://www.splunk.com/, 2021.

[44] XpoLog: Log management & analysis automations. http://www.xplg.com, 2021.

[45] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review*, 37(5):74–89, 2003.

[46] Michal Aharon, Gilad Barash, Ira Cohen, and Eli Mordechai. One graph is worth a thousand logs: Uncovering hidden structures in massive system event logs. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 227–243. Springer, 2009.

[47] Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. A systematic review on code clone detection. *IEEE access*, 7:86121–86144, 2019.

[48] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 281–293, 2014.

[49] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49, 2015.

[50] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

[51] Miltiadis Allamanis and Charles Sutton. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 472–483, 2014.

[52] Mohamed Aly. Survey on multiclass classification methods. *Neural Netw*, 19:1–9, 2005.

[53] James H Andrews. Testing using log file analysis: tools, methods, and issues. In *Proceedings 13th IEEE International Conference on Automated Software Engineering (Cat. No. 98EX239)*, pages 157–166. IEEE, 1998.

[54] Han Anu, Jie Chen, Wenchang Shi, Jianwei Hou, Bin Liang, and Bo Qin. An approach to recommendation of verbosity log levels based on logging intention. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 125–134. IEEE, 2019.

[55] Venera Arnaoudova, Sonia Haiduc, Andrian Marcus, and Giuliano Antoniol. The use of text retrieval and natural language processing in software engineering. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 949–950, 2015.

[56] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[57] Mahmoud Awad and Daniel A Menascé. Performance model derivation of operational systems through log analysis. In *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 159–168. IEEE, 2016.

[58] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[59] Dzmitry Bahdanau, Jan Chorowski, Dmitriy Serdyuk, Philemon Brakel, and Yoshua Bengio. End-to-end attention-based large vocabulary speech recognition. In *2016 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 4945–4949. IEEE, 2016.

[60] Liang Bao, Qian Li, Peiyao Lu, Jie Lu, Tongxiao Ruan, and Ke Zhang. Execution anomaly detection in large-scale systems through console log analysis. *Journal of Systems and Software*, 143:172–186, 2018.

[61] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: Online modelling and performance-aware systems. In *HotOS*, pages 85–90, 2003.

[62] Titus Barik, Robert DeLine, Steven Drucker, and Danyel Fisher. The bones of the system: A case study of logging and telemetry at Microsoft. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 92–101. IEEE, 2016.

[63] Victor R Basili, Forrest Shull, and Filippo Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, 1999.

[64] Roman Beck, Jacob Stenum Czepluch, Nikolaj Lollike, and Simon Malone. Blockchain–the gateway to trust-free cryptographic transactions. 2016.

[65] Christophe Bertero, Matthieu Roy, Carla Sauvanaud, and Gilles Trédan. Experience Report: Log Mining using Natural Language Processing and Application to Anomaly Detection. In *28th International Symposium on Software Reliability Engineering (ISSRE 2017)*, page 10p., Toulouse, France, October 2017.

[66] Avishkar Bhoopchand, Tim Rocktäschel, Earl Barr, and Sebastian Riedel. Learning python code suggestion with a sparse pointer network. *arXiv preprint arXiv:1611.08307*, 2016.

[67] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.", 2009.

[68] Peter Bodik, Moises Goldszmidt, Armando Fox, Dawn B Woodard, and Hans Andersen. Fingerprinting the datacenter: automated classification of performance crises. In *Proceedings of the 5th European conference on Computer systems*, pages 111–124. ACM, 2010.

[69] Jakub Breier and Jana Branišová. A dynamic rule creation based anomaly detection method for identifying security breaches in log records. *Wireless Personal Communications*, 94(3):497–511, 2017.

[70] John S Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In *Neurocomputing*, pages 227–236. Springer, 1990.

[71] Michael W Browne. Cross-validation methods. *Journal of mathematical psychology*, 44(1):108–132, 2000.

[72] J. Cândido, J. Haesen, M. Aniche, and A. van Deursen. An exploratory study of log placement recommendation in an enterprise system. In *2021 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 143–154, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[73] Jeanderson Candido, Maurício Aniche, and Arie van Deursen. Contemporary software monitoring: A systematic literature review. *arXiv preprint arXiv:1912.05878*, 2019.

[74] Songfei Cao, Xiaoqiang Di, Yue Gong, Weiwu Ren, and Xingxu Zhang. Knowledge extraction and knowledge graph construction based on campus security logss. In *International Conference on Artificial Intelligence and Security*, pages 354–364. Springer, 2020.

[75] Rich Caruana, Steve Lawrence, and C Lee Giles. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *Advances in neural information processing systems*, pages 402–408, 2001.

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

[77] Boyuan Chen and Zhen Ming Jiang. A survey of software log instrumentation. *ACM Computing Surveys (CSUR)*, 54(4):1–34, 2021.

[78] Boyuan Chen and Zhen Ming Jack Jiang. Characterizing and detecting anti-patterns in the logging code. In *Proceedings of the 39th International Conference on Software Engineering*, pages 71–81. IEEE Press, 2017.

[79] Boyuan Chen and Zhen Ming Jack Jiang. Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation. *Empirical Software Engineering*, 22(1):330–374, 2017.

[80] Boyuan Chen and Zhen Ming Jack Jiang. Extracting and studying the logging-code-issue-introducing changes in java-based large-scale open source software systems. *Empirical Software Engineering*, 24(4):2285–2322, 2019.

[81] Boyuan Chen and Zhen Ming Jack Jiang. Studying the use of java logging utilities in the wild. 2020.

[82] Boyuan Chen, Jian Song, Peng Xu, Xing Hu, and Zhen Ming Jack Jiang. An automated approach to estimating code coverage measures via execution logs. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 305–316. ACM, 2018.

[83] Mike Chen, Alice X Zheng, Jim Lloyd, Michael I Jordan, and Eric Brewer. Failure diagnosis using decision trees. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 36–43. IEEE, 2004.

[84] Rui Chen, Shenglin Zhang, Dongwen Li, Yuzhe Zhang, Fangrui Guo, Weibin Meng, Dan Pei, Yuzhi Zhang, Xu Chen, and Yuqing Liu. Logtransfer: Cross-system log anomaly detection for software systems with transfer learning. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 37–47. IEEE, 2020.

[85] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 217–231, 2014.

[86] Shaiful Chowdhury, Silvia Di Nardo, Abram Hindle, and Zhen Ming Jack Jiang. An exploratory study on assessing the energy impact of logging on android applications. *Empirical Software Engineering*, 23(3):1422–1456, 2018.

[87] Edward Chuah, Arshad Jhumka, Sai Narasimhamurthy, John Hammond, James C Browne, and Bill Barth. Linking resource usage anomalies with system failures from cluster log data. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, pages 111–120. IEEE, 2013.

[88] Edward Chuah, Shyh-hao Kuo, Paul Hiew, William-Chandra Tjhi, Gary Lee, John Hammond, Marek T Michalewicz, Terence Hung, and James C Browne. Diagnosing the root-causes of failures from cluster log files. In *2010 International Conference on High Performance Computing*, pages 1–10. IEEE, 2010.

[89] Kenneth W Church and William A Gale. A comparison of the enhanced good-turing and deleted estimation methods for estimating probabilities of english bigrams. *Computer Speech & Language*, 5(1):19–54, 1991.

[90] Marcello Cinque, Domenico Cotroneo, Roberto Natella, and Antonio Pecchia. Assessing and improving the effectiveness of logs for the analysis of software faults. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 457–466. IEEE, 2010.

[91] Marcello Cinque, Domenico Cotroneo, and Antonio Pecchia. Event logs for the analysis of software failures: A rule-based approach. *IEEE Transactions on Software Engineering*, 39(6):806–821, 2012.

[92] Raffaele Conforti, Marcello La Rosa, and Arthur HM ter Hofstede. Filtering out infrequent behavior from business process event logs. *IEEE Transactions on Knowledge and Data Engineering*, 29(2):300–314, 2016.

[93] Tim Cooijmans, Nicolas Ballas, César Laurent, Çağlar Gülçehre, and Aaron Courville. Recurrent batch normalization. *arXiv preprint arXiv:1603.09025*, 2016.

[94] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, Roberto Natella, and Nematollah Bidokhti. How bad can a bug get? an empirical analysis of software failures in the openstack cloud computing platform. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 200–211, 2019.

[95] Daniel Cukier. Devops patterns to scale web applications using cloud services. In *Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity*, pages 143–152, 2013.

[96] Hetong Dai, Heng Li, Che Shao Chen, Weiyi Shang, and Tse-Hsun Chen. Logram: Efficient log parsing using n-gram dictionaries. *IEEE Transactions on Software Engineering*, 2020.

[97] Subhasis Das and Chinmayee Shah. Contextual code completion using machine learning. Technical report, Technical Report. Stanford University, CA, USA, 2015.

[98] T. Davis and G. Shaw. *SQL Server Transaction Log Management*. Stairways handbook. Red Gate Books, 2012.

[99] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[100] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[101] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[102] Sheng Di, Hanqi Guo, Rinku Gupta, Eric R Pershey, Marc Snir, and Franck Cappello. Exploring properties and correlations of fatal events in a large-scale hpc system. *IEEE Transactions on Parallel and Distributed Systems*, 30(2):361–374, 2018.

[103] Sheng Di, Rinku Gupta, Marc Snir, Eric Pershey, and Franck Cappello. Logaider: A tool for mining potential correlations of hpc log events. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 442–451. IEEE, 2017.

[104] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. Log2: A cost-aware logging mechanism for performance diagnosis. In *USENIX Annual Technical Conference*, pages 139–150, 2015.

[105] Wei Dong, Luyao Luo, and Chao Huang. Dynamic logging with dylog in networked embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(1):1–25, 2015.

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

[107] 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, 2017.

[108] Andrej Dyck, Ralf Penners, and Horst Lichter. Towards definitions for release engineering and devops. In *2015 IEEE/ACM 3rd International Workshop on Release Engineering*, pages 3–3. IEEE, 2015.

[109] Lisa Ehrlinger and Wolfram Wöß. Towards a definition of knowledge graphs. *SEMANTiCS (Posters, Demos, SuCCESS)*, 48:1–4, 2016.

[110] Diana El-Masri, Fabio Petrillo, Yann-Gaël Guéhéneuc, Abdelwahab Hamou-Lhadj, and Anas Bouziane. A systematic literature review on automated log abstraction techniques. *Information and Software Technology*, 122:106276, 2020.

[111] Mostafa Farshchi, Jean-Guy Schneider, Ingo Weber, and John Grundy. Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pages 24–34. IEEE, 2015.

[112] Mostafa Farshchi, Jean-Guy Schneider, Ingo Weber, and John Grundy. Metric selection and anomaly detection for cloud operations using log and metric correlation analysis. *Journal of Systems and Software*, 137:531–549, 2018.

[113] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. X-trace: A pervasive network tracing framework. In *4th {USENIX} Symposium on Networked Systems Design & Implementation ({NSDI} 07)*, 2007.

[114] Elizabeth A Freeman and Gretchen G Moisen. A comparison of the performance of threshold criteria for binary classification in terms of predicted prevalence and kappa. *Ecological modelling*, 217(1-2):48–58, 2008.

[115] Ilenia Fronza, Alberto Sillitti, Giancarlo Succi, Mikko Terho, and Jelena Vlasenko. Failure prediction based on log files using random indexing and support vector machines. *Journal of Systems and Software*, 86(1):2–11, 2013.

[116] 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.

[117] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 24–33. ACM, 2014.

[118] Xiaoyu Fu, Rui Ren, Jianfeng Zhan, Wei Zhou, Zhen Jia, and Gang Lu. Logmaster: Mining event correlations in logs of large-scale cluster systems. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 71–80. IEEE, 2012.

[119] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 147–156, 2010.

[120] Evgeniy Gabrilovich and Shaul Markovitch. Wikipedia-based semantic interpretation for natural language processing. *Journal of Artificial Intelligence Research*, 34:443–498, 2009.

[121] Mohammad Gharehyazie, Baishakhi Ray, Mehdi Keshani, Masoumeh Soleimani Zavosht, Abbas Heydarnoori, and Vladimir Filkov. Cross-project code clones in github. *Empirical Software Engineering*, 24(3):1538–1573, 2019.

[122] Sina Gholamian. Leveraging Code Clones and Natural Language Processing for Log Statement Prediction. In *ASE '21: Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering - Doctoral Symposium*, November 2021.

[123] Sina Gholamian and Paul AS Ward. Logging statements' prediction based on source code clones. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 82–91, 2020.

[124] Sina Gholamian and Paul AS Ward. Borrowing from similar code: A deep learning nlp-based approach for log statement automation. *arXiv preprint arXiv:2112.01259*, 2021.

[125] Sina Gholamian and Paul AS Ward. A comprehensive survey of logging in software: From logging statements automation to log mining and analysis. *arXiv preprint arXiv:2110.12489*, 2021.

[126] Sina Gholamian and Paul AS Ward. On the naturalness and localness of software logs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 155–166. IEEE, 2021.

[127] Sina Gholamian and Paul AS Ward. What distributed systems say: A study of seven spark application logs. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 222–232. IEEE, 2021.

[128] Devarshi Ghoshal and Beth Plale. Provenance from log files: a bigdata problem. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pages 290–297, 2013.

[129] Andres Gomez, Lukas Sigrist, Thomas Schalch, Luca Benini, and Lothar Thiele. Efficient, long-term logging of rich data sensors using transient sensor nodes. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(1):1–23, 2017.

[130] Yvette Graham. Re-evaluating automatic summarization with bleu and 192 shades of rouge. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 128–137, 2015.

[131] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural networks*, 18(5-6):602–610, 2005.

[132] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2016.

[133] Haixuan Guo, Shuhan Yuan, and Xintao Wu. Logbert: Log anomaly detection via bert. *arXiv preprint arXiv:2103.04475*, 2021.

[134] Jin Guo, Mona Rahimi, Jane Cleland-Huang, Alexander Rasin, Jane Huffman Hayes, and Michael Vierhauser. Cold-start software analytics. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 142–153, 2016.

[135] Nentawe Gurumdimma, Arshad Jhumka, Maria Liakata, Edward Chuah, and James Browne. Crude: Combining resource usage data and error logs for accurate error detection in large-scale distributed systems. In *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, pages 51–60. IEEE, 2016.

[136] Apache Hadoop. A JIRA Issue for Hadoop. https://issues.apache.org/jira/browse/YARN-985, 2016.

[137] Sonia Haiduc, Venera Arnaoudova, Andrian Marcus, and Giuliano Antoniol. The use of text retrieval and natural language processing in software engineering. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 898–899, 2016.

[138] John L Hammond, Tommy Minyard, and Jim Browne. End-to-end framework for fault management for open source clusters: Ranger. In *Proceedings of the 2010 TeraGrid Conference*, pages 1–6, 2010.

[139] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. Logmine: Fast pattern recognition for log analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1573–1582, 2016.

[140] Ahmed Hassan, Daryl Martin, Parminder Flora, Paul Mansfield, and Dave Dietz. An industrial case study of customizing operational profiles using log compression. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 713–723. IEEE, 2008.

[141] Ahmed E Hassan. The road ahead for mining software repositories. In *2008 Frontiers of Software Maintenance*, pages 48–57. IEEE, 2008.

[142] Mehran Hassani, Weiyi Shang, Emad Shihab, and Nikolaos Tsantalis. Studying and detecting log-related issues. *Empirical Software Engineering*, 23(6):3248–3280, 2018.

[143] Gerhard Haßlinger and Oliver Hohlfeld. The gilbert-elliott model for packet loss in real time services on the internet. In *14th GI/ITG Conference-Measurement, Modelling and Evalutation of Computer and Communication Systems*, pages 1–15. VDE, 2008.

[144] Kimmo Hätönen, Jean François Boulicaut, Mika Klemettinen, Markus Miettinen, and Cyrille Masson. Comprehensive log compression with frequent patterns. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 360–370. Springer, 2003.

[145] Pinjia He, Zhuangbin Chen, Shilin He, and Michael R Lyu. Characterizing the natural language descriptions in software logging statements. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 178–189. IEEE, 2018.

[146] 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 *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 654–661. IEEE, 2016.

[147] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. Towards automated log parsing for large-scale log data analysis. *IEEE Transactions on Dependable and Secure Computing*, 15(6):931–944, 2017.

[148] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40. IEEE, 2017.

[149] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. Identifying impactful service system problems via log analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 60–70, 2018.

[150] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 207–218. IEEE, 2016.

[151] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. Loghub: a large collection of system log datasets towards automated log analytics. *arXiv preprint arXiv:2008.06448*, 2020.

[152] Kenneth Heafield. Kenlm: Faster and smaller language model queries. In *Proceedings of the sixth workshop on statistical machine translation*, pages 187–197, 2011.

[153] Kenneth Heafield, Ivan Pouzyrevsky, Jonathan H Clark, and Philipp Koehn. Scalable modified kneser-ney language model estimation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 690–696, 2013.

[154] David Heckerman, David Maxwell Chickering, Christopher Meek, Robert Rounthwaite, and Carl Kadie. Dependency networks for inference, collaborative filtering, and data visualization. *Journal of Machine Learning Research*, 1(Oct):49–75, 2000.

[155] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

[156] Abram Hindle, Alex Wilson, Kent Rasmussen, E Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. Greenminer: A hardware based mining software repositories software energy consumption framework. In *Proceedings of the 11th working conference on mining software repositories*, pages 12–21, 2014.

[157] Jonathan Ho, Nal Kalchbrenner, Dirk Weissenborn, and Tim Salimans. Axial attention in multidimensional transformers, 2019.

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

[159] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE, 2018.

[160] Shaohan Huang, Yi Liu, Carol Fung, Rong He, Yining Zhao, Hailong Yang, and Zhongzhi Luan. Hitanomaly: Hierarchical transformers for anomaly detection in system log. *IEEE Transactions on Network and Service Management*, 17(4):2064–2076, 2020.

[161] Yuan Huang, Xinyu Hu, Nan Jia, Xiangping Chen, Yingfei Xiong, and Zibin Zheng. Learning Code Context Information to Predict Comment Locations. *IEEE Transactions on Reliability*, 2019.

[162] Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez, and Erik Elmroth. Performance anomaly detection and bottleneck identification. *CSUR*, 48(1):1–35, 2015.

[163] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[164] John Irwin, Gregor Kickzales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, and J Loingtier. Aspect-oriented programming. *Proceedings of ECOOP, IEEE, Finland*, pages 220–242, 1997.

[165] Ramtin Jabbari, Nauman bin Ali, Kai Petersen, and Binish Tanveer. What is DevOps? A systematic mapping study on definitions and practices. In *Proceedings of the Scientific Workshop Proceedings of XP2016*, pages 1–11, 2016.

[166] Zhouyang Jia, Shanshan Li, Xiaodong Liu, Xiangke Liao, and Yunhuai Liu. Smartlog: Place error log statement by deep understanding of log intention. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 61–71. IEEE, 2018.

[167] Zhen Ming Jiang, Ahmed E Hassan, Gilbert Hamann, and Parminder Flora. Automatic identification of load testing problems. In *2008 IEEE International Conference on Software Maintenance*, pages 307–316. IEEE, 2008.

[168] Dan Jurafsky. *Speech & language processing*. Pearson Education India, 2000.

[169] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E Hassan. Logging library migrations: a case study for the apache software foundation projects. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 154–164. IEEE, 2016.

[170] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, Mark D Syer, and Ahmed E Hassan. Examining the stability of logging statements. *Empirical Software Engineering*, 23(1):290–333, 2018.

[171] Suhas Kabinna, Weiyi Shang, Cor-Paul Bezemer, and Ahmed E Hassan. Examining the stability of logging statements. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 326–337. IEEE, 2016.

[172] Staffs Keele et al. Guidelines for performing systematic literature reviews in software engineering. Technical report, Citeseer, 2007.

[173] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, volume 4, pages 1942–1948. IEEE, 1995.

[174] Samina Khalid, Tehmina Khalil, and Shamila Nasreen. A survey of feature selection and feature extraction techniques in machine learning. In *2014 science and information conference*, pages 372–378. IEEE, 2014.

[175] Taeyoung Kim, Suntae Kim, Sooyong Park, and YoungBeom Park. Automatic recommendation to appropriate log levels. *Software: Practice and Experience*, 50(3):189–209, 2020.

[176] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *International Conference on Learning Representations*, 2020.

[177] Barbara Ann Kitchenham, David Budgen, and Pearl Brereton. *Evidence-based software engineering and systematic reviews*, volume 4. CRC press, 2015.

[178] Egambaram Kodhai and Selvadurai Kanmani. Method-level code clone detection through lwh (light weight hybrid) approach. *Journal of Software Engineering Research and Development*, 2(1):12, 2014.

[179] Philipp Koehn. *Statistical machine translation*. Cambridge University Press, 2009.

[180] Robert V Krejcie and Daryle W Morgan. Determining sample size for research activities. *Educational and psychological measurement*, 30(3):607–610, 1970.

[181] Christopher Kruegel and Giovanni Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 251–261, 2003.

[182] William H Kruskal and W Allen Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, 47(260):583–621, 1952.

[183] Marcin Kubacki and Janusz Sosnowski. Holistic processing and exploring event logs. In *International Workshop on Software Engineering for Resilient Systems*, pages 184–200. Springer, 2017.

[184] Sangeeta Lal, Neetu Sardana, and Ashish Sureka. Logoptplus: Learning to optimize logging in catch and if programming constructs. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 215–220. IEEE, 2016.

[185] Sangeeta Lal, Neetu Sardana, and Ashish Sureka. Logging analysis and prediction in open source java project. In *Optimizing Contemporary Application and Processes in Open Source Software*, pages 57–85. IGI Global, 2018.

[186] Sangeeta Lal and Ashish Sureka. Logopt: Static feature extraction from source code for automated catch block logging prediction. In *Proceedings of the 9th India Software Engineering Conference*, pages 151–155. ACM, 2016.

[187] George Lee, Jimmy Lin, Chuang Liu, Andrew Lorek, and Dmitriy Ryaboy. The unified logging infrastructure for data analytics at twitter. *Proceedings of the VLDB Endowment*, 5(12):1771–1780, 2012.

[188] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. LogGC: garbage collecting audit log. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1005–1016, 2013.

[189] Mouad Lemoudden and Bouabid El Ouahidi. Managing cloud-generated logs using big data technologies. In *2015 International Conference on Wireless Networks and Mobile Communications (WINCOM)*, pages 1–7. IEEE, 2015.

[190] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, (8):707–710, 1966.

[191] Heng Li, Tse-Hsun Peter Chen, Weiyi Shang, and Ahmed E Hassan. Studying software logging using topic models. *Empirical Software Engineering*, pages 1–40, 2018.

[192] Heng Li, Weiyi Shang, Bram Adams, Mohammed Sayagh, and Ahmed E Hassan. A qualitative study of the benefits and costs of logging from developers' perspectives. *IEEE Transactions on Software Engineering*, 2020.

[193] Heng Li, Weiyi Shang, and Ahmed E Hassan. Which log level should developers choose for a new logging statement? *Empirical Software Engineering*, 22(4):1684–1716, 2017.

[194] Heng Li, Weiyi Shang, Ying Zou, and Ahmed E Hassan. Towards just-in-time suggestions for log changes. *Empirical Software Engineering*, 22(4):1831–1865, 2017.

[195] Ling Li and Hsuan-Tien Lin. Ordinal regression by extended binary classification. 2007.

[196] Shanshan Li, Xu Niu, Zhouyang Jia, Xiangke Liao, Ji Wang, and Tao Li. Guiding log revisions by learning from software evolution history. *Empirical Software Engineering*, pages 1–39, 2019.

[197] Shanshan Li, Xu Niu, Zhouyang Jia, Ji Wang, Haochen He, and Teng Wang. Logtracker: learning log revision behaviors proactively from software evolution history. In *Proceedings of the 26th Conference on Program Comprehension*, pages 178–188, 2018.

[198] Tao Li, Yexi Jiang, Chunqiu Zeng, Bin Xia, Zheng Liu, Wubai Zhou, Xiaolong Zhu, Wentao Wang, Liang Zhang, Jun Wu, et al. FLAP: An end-to-end event log analysis platform for system management. In *Proceedings of the 23rd ACM SIGKDD*

*International Conference on Knowledge Discovery and Data Mining*, pages 1547–1556, 2017.

[199] Tao Li, Wei Peng, Charles Perng, Sheng Ma, and Haixun Wang. An integrated data-driven framework for computing system management. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 40(1):90–99, 2010.

[200] Zhenhao Li, Tse-Hsun Chen, and Weiyi Shang. Where shall we log? studying and suggesting logging locations in code blocks. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 361–372. IEEE, 2020.

[201] Zhenhao Li, Tse-Hsun Chen, Jinqiu Yang, and Weiyi Shang. Dlfinder: Characterizing and detecting duplicate logging code smells. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 152–163. IEEE, 2019.

[202] Zhenhao Li, Tse-Hsun Peter Chen, Jinqiu Yang, and Weiyi Shang. Studying duplicate logging statements and their relationships with code clones. *IEEE Transactions on Software Engineering*, 2021.

[203] Zhenhao Li, Heng Li, Tse-Hsun Peter Chen, and Weiyi Shang. DeepLV: Suggesting Log Levels Using Ordinal Based Neural Networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1461–1472. IEEE, 2021.

[204] Yinglung Liang, Yanyong Zhang, Hui Xiong, and Ramendra Sahoo. Failure prediction in ibm bluegene/l event logs. In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*, pages 583–588. IEEE, 2007.

[205] Meng-Hui Lim, Jian-Guang Lou, Hongyu Zhang, Qiang Fu, Andrew Beng Jin Teoh, Qingwei Lin, Rui Ding, and Dongmei Zhang. Identifying recurrent and unknown performance issues. In *2014 IEEE International Conference on Data Mining*, pages 320–329. IEEE, 2014.

[206] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. *Text Summarization Branches Out*, 2004.

[207] 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.

[208] Jinyang Liu, Jieming Zhu, Shilin He, Pinjia He, Zibin Zheng, and Michael R Lyu. Logzip: Extracting Hidden Structures via Iterative Clustering for Log Compression. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 863–873. IEEE, 2019.

[209] Zhongxin Liu, Xin Xia, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. Which variables should i log? *IEEE Transactions on Software Engineering*, 2019.

[210] Steven Locke, Heng Li, Tse-Hsun Peter Chen, Weiyi Shang, and Wei Liu. LogAssist: Assisting Log Analysis Through Log Summarization. *IEEE Transactions on Software Engineering*, 2021.

[211] Dionysios Logothetis, Chris Trezzo, Kevin C Webb, and Kenneth Yocum. In-situ mapreduce for log processing. 2011.

[212] 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*, pages 23–25, 2010.

[213] Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. A neural architecture for generating natural language descriptions from source code changes. *arXiv preprint arXiv:1704.04856*, 2017.

[214] Siyang Lu, BingBing Rao, Xiang Wei, Byungchul Tak, Long Wang, and Liqiang Wang. Log-based abnormal task detection and root cause analysis for spark. In *ICWS*, pages 389–396. IEEE, 2017.

[215] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[216] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[217] Adetokunbo A.O. 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*, KDD '09, pages 1255–1264, New York, NY, USA, 2009. ACM.

[218] Allen D. Malony and et al. Performance measurement intrusion and perturbation analysis. *IEEE Computer Architecture Letters*, 3(04):433–450, 1992.

[219] José B Marino, Rafael E Banchs, Josep M Crego, Adrià de Gispert, Patrik Lambert, José AR Fonollosa, and Marta R Costa-jussà. N-gram-based machine translation. *Computational linguistics*, 32(4):527–549, 2006.

[220] Mark Marron. Log++ logging for a cloud-native world. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*, pages 25–36, 2018.

[221] Raffael Marty. Cloud application logging for forensics. In *proceedings of the 2011 ACM Symposium on Applied Computing*, pages 178–184, 2011.

[222] Ilias Mavridis and Helen Karatza. Performance evaluation of cloud-based log file analysis with apache hadoop and apache spark. *Journal of Systems and Software*, 125:133–151, 2017.

[223] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[224] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs. In *IJCAI*, volume 19, pages 4739–4745, 2019.

[225] Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. A search-based approach for accurate identification of log message formats. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 167–16710. IEEE, 2018.

[226] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael Rung-Tsong Lyu, and Hua Cai. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1245–1255, 2013.

[227] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[228] Andriy Miranskyy, Abdelwahab Hamou-Lhadj, Enzo Cialini, and Alf Larsson. Operational-log analysis for big data systems: Challenges and solutions. *IEEE Software*, 33(2):52–59, 2016.

[229] Tsuyoshi Mizouchi, Kazumasa Shimari, Takashi Ishio, and Katsuro Inoue. Padla: a dynamic log level adapter using online phase detection. In *Proceedings of the 27th International Conference on Program Comprehension*, pages 135–138. IEEE Press, 2019.

[230] Masayoshi Mizutani. Incremental mining of system log format. In *2013 IEEE International Conference on Services Computing*, pages 595–602. IEEE, 2013.

[231] Mordechai Mushkin and Israel Bar-David. Capacity and coding for the Gilbert-Elliott channels. *IEEE Transactions on Information Theory*, 35(6):1277–1290, 1989.

[232] Meiyappan Nagappan and Mladen A Vouk. Abstracting log lines to log event types for mining software system logs. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 114–117. IEEE, 2010.

[233] Meiyappan Nagappan, Kesheng Wu, and Mladen A Vouk. Efficiently extracting operational profiles from execution logs using suffix arrays. In *2009 20th International Symposium on Software Reliability Engineering*, pages 41–50. IEEE, 2009.

[234] 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.

[235] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*, 2010.

[236] Animesh Nandi, Atri Mandal, Shubham Atreja, Gargi B Dasgupta, and Subhrajit Bhattacharya. Anomaly detection using program control flow graph mining from execution logs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 215–224, 2016.

[237] Bianca Napoleão, Katia Romero Felizardo, Érica Ferreira de Souza, and Nandamudi L Vijaykumar. Practical similarities and differences between systematic literature reviews and systematic mappings: a tertiary study. In *SEKE*, pages 85–90, 2017.

[238] Sasho Nedelkoski, Jasmin Bogatinovski, Alexander Acker, Jorge Cardoso, and Odej Kao. Self-supervised log parsing. *arXiv preprint arXiv:2003.07905*, 2020.

[239] Andrew Y Ng et al. Preventing" overfitting" of cross-validation data. In *ICML*, volume 97, pages 245–253. Citeseer, 1997.

[240] David Ogle, Heather Kreger, Abdi Salahshour, Jason Cornpropst, Eric Labadie, Mandy Chessell, Bill Horn, John Gerken, James Schoech, and Mike Wamboldt. Canonical situation data format: The common base event v1. 0.1. *International Business Machines Corporation*, 2004.

[241] Adam Oliner and Jon Stearley. What supercomputers say: A study of five system logs. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 575–584. IEEE, 2007.

[242] Adam J Oliner and Alex Aiken. Online detection of multi-component interactions in production systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 49–60. IEEE, 2011.

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

[244] Alina Oprea, Zhou Li, Ting-Fang Yen, Sang H Chin, and Sumayah Alrwais. Detection of early-stage enterprise infection by mining large-scale log data. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 45–56. IEEE, 2015.

[245] Andrew J Page, Thomas M Keane, and Thomas J Naughton. Multi-heuristic dynamic task allocation using genetic algorithms in a heterogeneous distributed system. *JPDC*, 70(7):758–766, 2010.

[246] Nikhil R Pal and Sankar K Pal. Entropy: A new definition and its applications. *IEEE transactions on systems, man, and cybernetics*, 21(5):1260–1270, 1991.

[247] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

[248] Sungheon Park and Nojun Kwak. Analysis on the dropout effect in convolutional neural networks. In *Asian conference on computer vision*, pages 189–204. Springer, 2016.

[249] Ronald K Pearson, Yrjö Neuvo, Jaakko Astola, and Moncef Gabbouj. Generalized hampel filters. *EURASIP Journal on Advances in Signal Processing*, 2016(1):1–18, 2016.

[250] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo. Industry practices and event logging: Assessment of a critical software development process. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 169–178, May 2015.

[251] Antonio Pecchia and Stefano Russo. Detection of software failures through event logs: An experimental study. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 31–40. IEEE, 2012.

[252] Jeffrey Pennington, Richard Socher, and Christopher D Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[253] Chang-Shing Perng, David Thoenen, Genady Grabarnik, Sheng Ma, and Joseph Hellerstein. Data-driven validation, completion and construction of event relationship networks. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 729–734, 2003.

[254] Panagiotis Petridis, Anastasios Gounaris, and Jordi Torres. Spark parameter tuning via trial-and-error. In *INNS Conference on Big Data*, pages 226–237. Springer, 2016.

[255] Thomas Plötz and Gernot A Fink. Markov models for offline handwriting recognition: a survey. *International Journal on Document Analysis and Recognition (IJDAR)*, 12(4):269, 2009.

[256] David MW Powers. Applications and explanations of zipf's law. In *New methods in language processing and computational natural language learning*, 1998.

[257] David MW Powers. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. *arXiv preprint arXiv:2010.16061*, 2020.

[258] Tongqing Qiu, Zihui Ge, Dan Pei, Jia Wang, and Jun Xu. What happened in my network: mining network events from router syslogs. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 472–484, 2010.

[259] Ariel Rabkin, Wei Xu, Avani Wildani, Armando Fox, David A Patterson, and Randy H Katz. A graphical representation for identifier structure in logs. In *SLAML*, 2010.

[260] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

[261] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[262] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–428, 2014.

[263] Eric Sven Ristad and Peter N Yianilos. Learning string-edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):522–532, 1998.

[264] Fanny Rivera-Ortiz and Liliana Pasquale. Towards automated logging for forensic-ready software systems. In *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW)*, pages 157–163. IEEE, 2019.

[265] Guoping Rong, Yangchen Xu, Shenghui Gu, He Zhang, and Dong Shao. Can You Capture Information As You Intend To? A Case Study on Logging Practice in Industry. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 12–22. IEEE, 2020.

[266] Chanchal K Roy, Minhaz F Zibran, and Rainer Koschke. The vision of software clone management: Past, present, and future (keynote paper). In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 18–33. IEEE, 2014.

[267] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. 2007.

[268] Alexander M Rush, Sumit Chopra, and Jason Weston. A neural attention model for abstractive sentence summarization. *arXiv preprint arXiv:1509.00685*, 2015.

[269] Barbara Russo, Giancarlo Succi, and Witold Pedrycz. Mining system logs to learn error predictors: a case study of a telemetry system. *Empirical Software Engineering*, 20(4):879–927, 2015.

[270] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 354–365, 2018.

[271] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: scaling code clone detection to big-code. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 1157–1168. IEEE, 2016.

[272] Felix Salfner, Maren Lenk, and Miroslaw Malek. A survey of online failure prediction methods. *ACM Computing Surveys (CSUR)*, 42(3):10, 2010.

[273] Felix Salfner, Steffen Tschirpke, and Miroslaw Malek. Comprehensive logfiles for autonomic systems. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 211. IEEE, 2004.

[274] Mohammed Salman, Brian Welch, David Raymond, Randy Marchany, and Joseph Tront. Designing phelkstat: Big data analytics for system event logs. HICSS Symposium on Cybersecurity Big Data Analytics, 2017.

[275] Ioannis Samoladas, Georgios Gousios, Diomidis Spinellis, and Ioannis Stamelos. The sqo-oss quality model: measurement based open source software evaluation. In *IFIP International Conference on Open Source Systems*, pages 237–248. Springer, 2008.

[276] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[277] Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E Hassan, Michael W Godfrey, Mohamed Nasser, and Parminder Flora. An exploratory study of the evolution of communicated information about the execution of large software systems. *Journal of Software: Evolution and Process*, 26(1):3–26, 2014.

[278] Weiyi Shang, Zhen Ming Jiang, Hadi Hemmati, Brain Adams, Ahmed E Hassan, and Patrick Martin. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 402–411. IEEE, 2013.

[279] Weiyi Shang, Meiyappan Nagappan, and Ahmed E Hassan. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, 20(1):1–27, 2015.

[280] Weiyi Shang, Meiyappan Nagappan, Ahmed E Hassan, and Zhen Ming Jiang. Understanding log lines using development knowledge. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 21–30. IEEE, 2014.

[281] Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE mobile computing and communications review*, 5(1):3–55, 2001.

[282] Yun Shen, Enrico Mariconti, Pierre Antoine Vervier, and Gianluca Stringhini. Tiresias: Predicting security events through deep learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 592–605, 2018.

[283] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proceedings of the VLDB Endowment*, 8(13):2110–2121, 2015.

[284] Keiichi Shima. Length matters: Clustering system log messages using length of words. *arXiv preprint arXiv:1611.03213*, 2016.

[285] Donghwan Shin, Domenico Bianculli, and Lionel Briand. Effective removal of operational log messages: an application to model inference. *arXiv preprint arXiv:2004.07194*, 2020.

[286] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. 2010.

[287] Przemysław Skibiński and Jakub Swacha. Fast and efficient log file compression. In *proceedings of 11th east-European conference on advances in databases and information systems (ADBIS)*, pages 330–342, 2007.

[288] Niyazi Sorkunlu, Varun Chandola, and Abani Patra. Tracking system behavior from resource usage data. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 410–418. IEEE, 2017.

[289] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[290] Xiaoxiao Sun, Wenjie Hou, Dongjin Yu, Jiaojiao Wang, and Jianliang Pan. Filtering out noise logs for process modelling based on event dependency. In *2019 IEEE International Conference on Web Services (ICWS)*, pages 388–392. IEEE, 2019.

[291] Mark D Syer, Zhen Ming Jiang, Meiyappan Nagappan, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. Leveraging performance counters and execution logs to

diagnose memory-related performance issues. In *2013 IEEE international conference on software maintenance*, pages 110–119. IEEE, 2013.

[292] Narate Taerat, Jim Brandt, Ann Gentile, Matthew Wong, and Chokchai Leangsuksun. Baler: deterministic, lossless log message clustering tool. *Computer Science-Research and Development*, 26(3):285–295, 2011.

[293] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* iComment: Bugs or bad comments?*. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 145–158, 2007.

[294] Liang Tang, Tao Li, and Chang-Shing Perng. Logsig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 785–794. ACM, 2011.

[295] Liang Tang, Tao Li, and Chang-Shing Perng. LogSig: Generating System Events from Raw Textual Logs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 785–794, New York, NY, USA, 2011. ACM.

[296] Ran Tang and Ying Zou. An approach for mining web service composition patterns from execution logs. In *2010 12th IEEE International Symposium on Web Systems Evolution (WSE)*, pages 53–62. IEEE, 2010.

[297] JavaParser Development Team. Java parser. https://github.com/javaparser/javaparser, 2019.

[298] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280, 2014.

[299] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)*, pages 119–126. IEEE, 2003.

[300] Risto Vaarandi and Mauno Pihelgas. Logcluster-A data clustering and pattern mining algorithm for event logs. In *Network and Service Management (CNSM), 2015 11th International Conference on*, pages 1–7. IEEE, 2015.

[301] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[302] Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. Context-aware retrieval-based deep commit message generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(4):1–30, 2021.

[303] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271. IEEE, 2020.

[304] Wenshuo Wang and Ding Zhao. Extracting traffic primitives directly from naturalistically logged data for self-driving applications. *IEEE Robotics and Automation Letters*, 3(2):1223–1229, 2018.

[305] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. Retrieve and refine: exemplar-based neural comment generation. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 349–360. IEEE, 2020.

[306] Bernard L Welch. The generalization ofstudent's' problem when several different population variances are involved. *Biometrika*, 34(1/2):28–35, 1947.

[307] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 334–345. IEEE, 2015.

[308] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, pages 1–10, 2014.

[309] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[310] Rongxin Wu, Xiao Xiao, Shing-Chi Cheung, Hongyu Zhang, and Charles Zhang. Casper: An efficient approach to call trace collection. *ACM SIGPLAN Notices*, 51(1):678–690, 2016.

[311] 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.

[312] Wei Xu and Alex Rudnicky. Can artificial neural networks learn language models? In *Sixth international conference on spoken language processing*, 2000.

[313] Xiwei Xu, Ingo Weber, Len Bass, Liming Zhu, Hiroshi Wada, and Fei Teng. Detecting cloud provisioning errors using an annotated process model. In *Proceedings of the 8th Workshop on Middleware for Next Generation Internet Computing*, page 5. ACM, 2013.

[314] Xiwei Xu, Liming Zhu, Ingo Weber, Len Bass, and Daniel Sun. POD-Diagnosis: Error diagnosis of sporadic operations on cloud applications. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 252–263. IEEE, 2014.

[315] Lin Yang, Junjie Chen, Zan Wang, Weijing Wang, Jiajun Jiang, Xuyuan Dong, and Wenbin Zhang. Semi-supervised log-based anomaly detection via probabilistic label estimation. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1448–1460. IEEE, 2021.

[316] Stephen Yang, Seo Jin Park, and John Ousterhout. Nanolog: a nanosecond scale logging system. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 335–350, 2018.

[317] Kundi Yao, Guilherme B de Pádua, Weiyi Shang, Steve Sporea, Andrei Toma, and Sarah Sajedi. Log4Perf: Suggesting Logging Locations for Web-based Systems' Performance Monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 127–138. ACM, 2018.

[318] Kundi Yao, Guilherme B de Padua, Weiyi Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi. Log4perf: suggesting and updating logging locations for web-based systems' performance monitoring. *Empirical Software Engineering*, 25(1):488–531, 2020.

[319] Kundi Yao, Heng Li, Weiyi Shang, and Ahmed E Hassan. A study of the performance of general compressors on log files. *Empirical Software Engineering*, pages 3043–3085, 2020.

[320] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. Neural detection of semantic code clones via tree-based convolution. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 70–80. IEEE, 2019.

[321] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. 44(2):489–502, 2016.

[322] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 249–265, 2014.

[323] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *ACM SIGARCH computer architecture news*, volume 38, pages 143–154. ACM, 2010.

[324] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: enhancing failure diagnosis with proactive logging. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 293–306, 2012.

[325] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering*, pages 102–112. IEEE Press, 2012.

[326] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)*, 30(1):4, 2012.

[327] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th {USENIX} ({NSDI} 12)*, pages 15–28, 2012.

[328] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. Spark: Cluster computing with working sets.

[329] Lei Zeng, Yang Xiao, and Hui Chen. Linux auditing: overhead and adaptation. In *2015 IEEE International Conference on Communications (ICC)*, pages 7168–7173. IEEE, 2015.

[330] Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun Peter Chen. Studying the characteristics of logging practices in mobile apps: a case study on f-droid. *Empirical Software Engineering*, pages 1–41, 2019.

[331] ChengXiang Zhai and Sean Massung. *Text data management and analysis: a practical introduction to information retrieval and text mining*. Association for Computing Machinery and Morgan & Claypool, 2016.

[332] Bo Zhang, Hongyu Zhang, Pablo Moscato, and Aozhong Zhang. Anomaly detection via mining numerical workflow relations from logs. In *2020 International Symposium on Reliable Distributed Systems (SRDS)*, pages 195–204. IEEE, 2020.

[333] Cheng Zhang, Zhenyu Guo, Ming Wu, Longwen Lu, Yu Fan, Jianjun Zhao, and Zheng Zhang. Autolog: facing log redundancy and insufficiency. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 10. ACM, 2011.

[334] Ke Zhang, Jianwu Xu, Martin Renqiang Min, Guofei Jiang, Konstantinos Pelechrinis, and Hui Zhang. Automated it system failure prediction: A deep learning approach. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 1291–1300. IEEE, 2016.

[335] Min Zhang, Tracy Hall, and Nathan Baddoo. Code bad smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: research and practice*, 23(3):179–202, 2011.

[336] Steve Zhang, Ira Cohen, Moises Goldszmidt, Julie Symons, and Armando Fox. Ensembles of models for automated diagnosis of system performance problems. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 644–653. IEEE, 2005.

[337] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 807–817, 2019.

[338] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 19–33, 2017.

[339] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 565–581, New York, NY, USA, 2017. ACM.

[340] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 603–618, 2016.

[341] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 629–644, 2014.

[342] Chen Zhi, Jianwei Yin, Shuiguang Deng, Maoxin Ye, Min Fu, and Tao Xie. An exploratory study of logging configuration practice in java. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 459–469. IEEE, 2019.

[343] Pengpeng Zhou, Yang Wang, Zhenyu Li, Xin Wang, Gareth Tyson, and Gaogang Xie. LogSayer: Log Pattern-driven Cloud Component Anomaly Diagnosis with Machine Learning. In *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2020.

[344] Qingyu Zhou, Nan Yang, Furu Wei, Shaohan Huang, Ming Zhou, and Tiejun Zhao. Neural document summarization by jointly learning to score and select sentences. *arXiv preprint arXiv:1807.02305*, 2018.

[345] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of FSE/ESCE*, pages 683–694, 2019.

[346] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. Learning to log: Helping developers make informed logging decisions. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 415–425. IEEE Press, 2015.

[347] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st*

*International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 121–130. IEEE, 2019.

[348] George Kingsley Zipf. *Human behavior and the principle of least effort: An introduction to human ecology*. Ravenio Books, 2016.

[349] De-Qing Zou, Hao Qin, and Hai Jin. Uilog: Improving log-based fault diagnosis by log analysis. *Journal of computer science and technology*, 31(5):1038–1052, 2016.

# Chapter 11

# Summary of Publications

This research has contributed to the following publications which are either, *published*, *accepted*, or *under review*. I was the Principal Investigator and first author of all co-authored contributions.

- The systematic literature review that I have conducted in Chapter 2 is in the *second review phase* in *IEEE Journal of Transactions on Software Engineering* [125].
- My work in Chapter 3 for clone-based log prediction has been accepted in *2021 36th IEEE/ACM International Conference on Automated Software Engineering, Doctoral Symposium (ASE)* [122].
- My research on log location prediction in Chapter 5 has been published in *2020 35th ACM/SIGAPP Symposium On Applied Computing (ACM SAC)* [123].
- My work on log description prediction in Chapter 6 is under review [124].
- My research in Chapter 8 for NLP attributes of log files is published in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)* [126].
- My study in Chapter 7 on logging cost and benefit is published in *2021 The 40th International Symposium on Reliable Distributed Systems (SRDS)* [127].