JLLAR: A Logging Recommendation Plug-in Tool for Java

Jing Zhu*
Software Institute, Nanjing University
Nanjing, Jiangsu, China
151250211@smail.nju.edu.cn

Guoping Rong
Software Institute, Nanjing University
Nanjing, Jiangsu, China
ronggp@nju.edu.cn

Guocheng Huang
Software Institute, Nanjing University
Nanjing, Jiangsu, China
orihgc02221@gmail.com

Shenghui Gu
Software Institute, Nanjing University
Nanjing, Jiangsu, China
dz1732002@smail.nju.edu.cn

He Zhang
Software Institute, Nanjing University
Nanjing, Jiangsu, China
hezhang@nju.edu.cn

Dong Shao
Software Institute, Nanjing University
Nanjing, Jiangsu, China
dongshao@nju.edu.cn

ABSTRACT

Logs are the execution results of logging statements in software systems after being triggered by various events, which is able to capture the dynamic behavior of software systems during runtime and provide important information for software analysis, e.g., issue tracking, performance monitoring, etc. Obviously, to meet this purpose, the quality of the logs is critical, which requires appropriately placement of logging statements. Existing research on this topic reveals that *where to log?* and *what to log?* are two most concerns when conducting logging practice in software development, which mainly relies on developers' personal skills, expertise and preference, rendering several problems impacting the quality of the logs inevitably. One of the reasons leading to this phenomenon might be that several recognized best practices(strategies as well) are easily neglected by software developers. Especially in those software projects with relatively large number of participants. To address this issue, we designed and implemented a plug-in tool (i.e., *JLLAR*) based on the Intellij IDEA, which applied machine learning technology to identify and create a set of rules reflecting commonly recognized logging practices. Based on this rule set, *JLLAR* can be used to scan existing source code to identify issues regarding the placement of logging statements. Moreover, *JLLAR* also provides automatic code completion and semi code completion (i.e., to provide recommendations) regarding logging practice to support software developers during coding.

KEYWORDS

logging practice, machine learning, tool

ACM Reference Format:

Jing Zhu, Guoping Rong, Guocheng Huang, Shenghui Gu, He Zhang, and Dong Shao. 2019. JLLAR: A Logging Recommendation Plug-in Tool for Java. In *Internetware '19: Proceedings of the 11th Asia-Pacific Symposium on Internetware (Internetware '19), October 28–29, 2019, Fukuoka, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3361242.3361261

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Internetware '19, October 28–29, 2019, Fukuoka, Japan

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7701- 0/19/10…$15.00
https://doi.org/10.1145/3361242.3361261

Table 1: The Terminologies Adopted In This Paper

Log	A sequence of log entries. Logs are usually stored in a textual file or a database with typical information such as timestamps, value of variables, recognizable text, etc.
Log Analysis	An art and science seeking to make sense out of logs. Analysts utilize various techniques to analyze the existing logs and retrieve useful information to support decisions
Logging Practice	A common programming practice in modern software development, typically issued by inserting logging statements in source code.
Logging Statement	The statements developers insert into source code in order to record runtime information of software systems when being triggered.
Logging Strategy	The decision on where to log and what to log.

1 INTRODUCTION

As software systems become more widely used, modern software systems are becoming more and more complicated and prone to evolution, rendering big challenges for software engineers to maintain these software systems[16]. Logging technology has attracted more and more attention since the resulted logs could be used to capture the runtime behavior of software systems or services, which provides valuable information for software maintenance. A variety of software engineering tasks with diverse purposes depend on logs, for example, debugging, monitoring, auditing, defect prediction and so on [11, 16, 17, 24, 27]. In particular, logs are extremely valuable for software developers and testers to diagnose failures both in testing environment and production environment [14, 19]. Sometimes it is the only way for software engineers to deal with production failures[19].

To facilitate elaboration and understanding, we first clarify several terminologies commonly adopted in this paper. As shown in Table 1.

Logging practice is a common programming practice in modern software development, typically issued by inserting logging statements in source code. The importance of logging practice has been widely recognized in industry [15]. In real-world production, software engineers need to record the error or warning information and the information indicating the success of an event as well so as to establish an understanding to the dynamic behavior of software systems. Apparently, to be useful, logs generated by logging statements in the source code should be well-formed and informative, which requires the corresponding logging practice to be carried out properly. However, it is normally difficult to make sound decisions to determine the context of logging statements (*where to log?*) and the content of logging statements (*what to log?*) [13, 23, 29].

Study [5] lists several scenarios, with which developers should not put logging statements. Meanwhile, the content of logging statement could also influence its capability to capture the runtime behavior of software systems. For example, study [12] reveals that more than half of logging statements even do not contain any variables. Other issues regarding the logging practices include *not using log statements, missing log statements at key locations, inappropriate use of log statement levels*, and *lack of parameter information*, etc.[12] As a matter of fact, a study based on open source project indicate that logging practice is usually not consistently conducted across the whole software system in most projects and some projects are extremely low-level in terms of the density of logging statements, rendering questionable satisfaction of the primary purpose of logging practice, i.e., to capture the dynamic behavior of software systems during runtime[25].

It seems that lacking of practical guidelines to carry out logging practice is one of the reasons for the above phenomenon. In order to address this problem, we designed and implemented a plug-in tool, namely *JLLAR* (Java Log Lint and Analysis Plug-in) to help software developers normalize logging practice and guarantee some well-recognized best practices on logging practice be fully implemented. To do this, a build-in rule set is applied in *JLLAR*, which is derived from some well-recognized best practice on logging practice. We also applied machine learning technology to identify actual context for logging statements in actual software projects so as to further refine the rule set.

The rest of this paper is organized as follows. The second section introduces some relevant research status on logging practice and supporting tools. Section III provides the design and implementation of the tool (i.e., *JLLAR*). Section IV shows some preliminary evaluation results of *JLLAR*. We discuss the implication of our study and the limitation as well in Section V. Finally, we conclude this paper in the last section with suggestions for future work.

2 RELATED WORK

The importance of logging practice in modern software engineering is widely recognized, which attracts a lot of attention among researchers. In this section, we present the current research status of logging practice.

2.1 Log Analysis

The majority of the researches on logging practice concentrates on log analysis [1], which means analysts utilize various techniques to analyze the existing logs and retrieve useful information to support decisions, e.g., causal paths [9], event correlations [14, 28], component dependency [22], resource usage [3], etc.

2.2 Logging Practices

Apparently, log with high quality is the prerequisite of useful log analysis, which requires logging practice being carried out properly. Nevertheless, compared to the large amount of studies on log analysis, there are fewer studies focusing on logging practice.

Yuan et al. conducted a series of studies on logging practice. For example, they confirmed the importance of logging practice with quantitative evidences [8]. In another study [9], researchers proposed an approach to enhance the existing logging statements to collect causally-related information so as to reduce the burden when diagnosing failures. In [7], Yuan et al. identified a number of patterns for logging practice to put logging statements so as to balance âĂIJover loggingâĂİ and âĂIJinsufficiently loggingâĂİ. To make logging practice more effective, they proposed to automate the placement of logging statements by measuring the degree of software uncertainty that can be removed by adding a logging statement [29]. The approach is able to compute an optimal logging statement placement, disambiguating the entire function call path with acceptable performance slowdown.

In spite of the aforementioned effort made by researchers, logging has been still an arbitrary and subjective practice in industry. There are no well-defined and broadly-accepted logging guidelines for developers to refer to during software development [2, 23], not to mention that some imperfect guidelines for logging practice can be fully implemented in real-world software projects[25].

3 THE DESIGN OF JLLAR

In this section, we provide the design and implementation of the tool. First, we introduce the process to establish a rule set which is derived from serveral well recongnized best pracitce regarding logging practice. Moreover, a machine learning technology is also applied to refine the rules to reflect their application context in pratice. Based on the rule set, a plug-in tool is proposed to support programming in Java language.

3.1 Basic rule set

There are several well recognized best practices to address issues related to logging practice in published literatures and technical blogs, e.g., not using logging statements, missing logging statements at key locations, inappropriate use of logging statement level, lack of parameter and defect introduction. To collect these practices as comprehensively as possible, we retrieved the literatures included in a SLR (Systematic Literature Review) study [26] and extracted the content related to either best practices or anti-patterns related to logging practice, as shown in the Figure **??**. Then we manually we evaluate, convert, and filter these extracted results and establish a basic rule set as shown in Table 2.

3.2 Refine rules

Basic rules provide a foundation for logging recommendation, however, real-world projects might take a different logging placement strategy in production environment, as mentioned in subsection **??**. Since there seems no simple solution to this issue, we adopted machine learning technology to study the situation and context in real-world software projects. Based on the model derived from machine learning, we refine the rule set behind *JLLAR* to recommend logging practice better. Figure 1 shows the steps to convert textual features to numeric textual features for the model building phase.

The main steps of this process are as follows:

(1) Use the Java parser to build an abstract syntax tree corresponding to the Java file, traversing access to each number node, accessing each logging code block.
(2) Use multiple regular expression filters to determine whether a code block has a logging statement, and paste the tag for the code block.

Table 2: Built-in rules

Number	Rule Description	Problem
1	Using logging statements instead of system.out or system.err	Not Using Logging Statements
2	Assertions need to be logged	Missing Logging Statements At Key Locations
3	Exception needs to be logged	Missing Logging Statements At Key Locations
4	Logical branches need to be logged	Missing Logging Statements At Key Locations
5	Assertions need to be logged	Missing Logging Statements At Key Locations
6	Logging in an exception should use ERROR or higher	Inappropriate Use of Logging Statement Level
7	Logging statement should contain at least one variable	Lack of Parameter

(3) Feature extraction, also using the Java parser for feature extraction, the obtained features are divided into three types, digital features, Boolean features and text features. Table 3 presents the features extracted in the previous section.
(4) Convert text features to digital text features through a text mining classifier
(5) Classify the preprocessed data to generate a model
(6) Finally, cross-project forecasting and evaluation of experimental results.

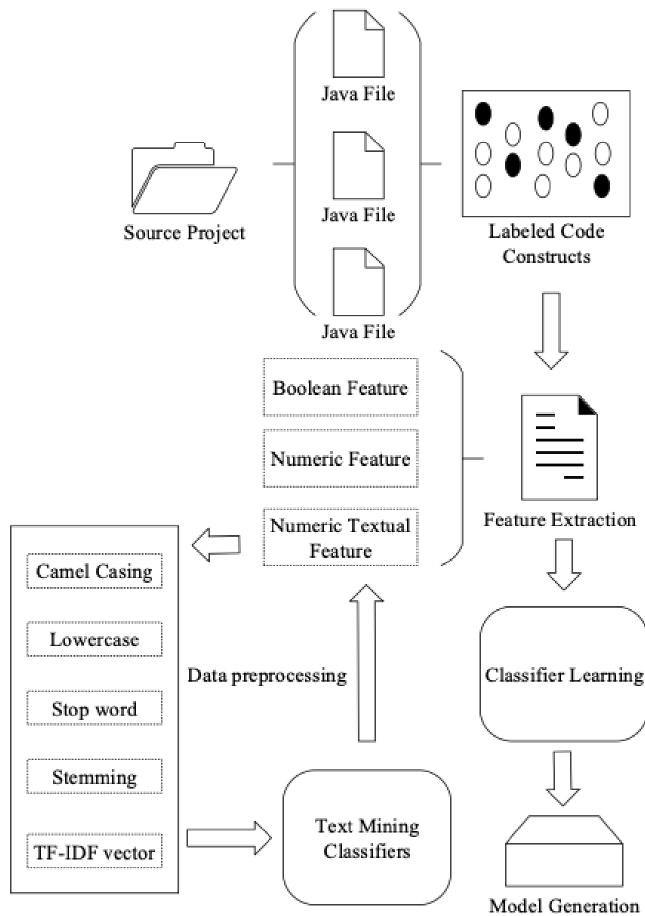

Figure 1: Process of Model Training

In the application, JLLAR extracts features from all the content that has been written, enters into the model and returns the results of the logging recommendation, as shown in Figure ??.

Table 3: Features

Number	Feature Description	Domain	Type
1	Size of Try Block	Try/Catch	Numeric
2	Size of Method BT	Method BT	Numeric
3	Catch Exception Type	Try/Catch	Textual
4	Previous Catch Block	Try/Catch	Boolean
5	Logged Previous Catch Block	Try/Catch	Boolean
6	Logged Try Block	Try/Catch	Boolean
7	Logged Method BT	Method BT	Boolean
8	Logging Count in Try Block	Try/Catch	Numeric
9	Logging Count in Method BT	Method BT	Numeric
10	Logging Level in Catch Block	Try/Catch	Textual
11	Logging Level in Method BT	Method BT	Textual
12	Operators in Try Block	Try/Catch	Textual
13	Operators in Method BT	Method BT	Boolean
14	Operator Count in Try Blocks	Try/Catch	Numeric
15	Operator Count Method BT	Method BT	Numeric
16	Variable Count in Try Block	Try/Catch	Numeric
17	Variable Count in Method BT	Method BT	Numeric
18	Method Call Count in Try Block	Try/Catch	Numeric
19	Method Call Count in Method BT	Method BT	Numeric
20	Container Method have Parameter	Other	Boolean
21	Container Method Parameter Count	Other	Boolean
22	Container Method Parameter Type	Other	Textual
23	Container Method Parameter Name	Other	Textual
24	If in Try	Try/Catch	Boolean
25	If in Method BT	Method BT	Boolean
26	If Count in Try	Try/Catch	Numeric
27	If Count in Method BT	Method BT	Numeric
28	Container Package Name	Other	Textual
29	Container Class Name	Other	Textual
30	Container Method Name	Other	Textual
31	Variable Name in Try Block	Try/Catch	Textual
32	Variable Name in Method BT	Method BT	Textual
33	Method Call Name in Try Block	Try/Catch	Textual
34	Method Call Name in Method BT	Method BT	Textual
35	Throw/Throws in Try Block	Try/Catch	Boolean
36	Throw/Throws in Catch Block	Try/Catch	Boolean
37	Throw/Throws in Method BT	Method BT	Boolean
38	Return in Try Block	Try/Catch	Boolean
39	Return in Catch Block	Try/Catch	Boolean
40	Return in Method BT	Method BT	Boolean
41	Assert in Try Block	Try/Catch	Boolean
42	Assert in Catch Block	Try/Catch	Boolean
43	Assert in Method BT	Method BT	Boolean
44	Thread.Sleep in Try Block	Try/Catch	Boolean
45	Interrupted Exception Type	Try/Catch	Boolean
46	Exception Object 'Ignore' in Catch	Try/Catch	Boolean

In the end, we get the results as shown in Table 4. It can be seen from the comparison results of the indicators that the prediction effects of the algorithms are not much different. Under the comprehensive comparison, the indicators of the random forest model are slightly superior, and the AUC index is significantly higher. Adaboost, also higher than the other three models, shows that its classification effect is the best.

3.3 Tool design

IntelliJ IDEA[1] is an integrated development environment for the Java programming language. It is recognized as one of the best

[1]https://www.jetbrains.com/idea/

Table 4: Evaluation of Indicator Results of Models

Algorithm	Prediction	Recall	F-measure	Accuracy	RA
Adaboost[10]	0.982	0.314	0.476	0.825	0.695
SVM[6]	0.979	0.311	0.472	0.824	0.769
Naive Bayes[20]	0.959	0.313	0.476	0.823	0.766
Random Forest[4]	0.987	0.317	0.476	0.825	0.786
Logistics Regression[21]	0.982	0.314	0.476	0.825	0.695

Java development tools in the industry, especially in intelligent code assistants, code prompts, refactoring, J2EE support, various version tools, JUnit, CVS integration, code analysis, innovative GUI design, etc. The function can be said to be extraordinary. RebelLabs conducted a survey of Java tools and technologies for 2016, with 46% of developers using IntelliJ IDEA, which has exceeded 41% of Eclipse[18].

Virtual File System. The Virtual File System is a component of the IntelliJ platform that encapsulates most of the activities used to process files. Its main uses include:

Program Structure Interface. The program structure interface is a layer in the IntelliJ platform that is responsible for parsing files and creating syntax and semantic code models that support so many platform features.

Component Persistence. Component Persistence is an API provided by the IntelliJ platform that allows components or services to restart their state between persistent IDEs. You can use a simple API to save some values, or use the PersistentStateComponent interface to save the state of more complex components.

3.3.1 Architecture. The architecture diagram of *JLLAR* is shown in the Figure 2, with a total of eight main modules. The GUI module is responsible for providing an interface to the user, including a log frame configuration interface, a built-in rule configuration interface, and a custom rule editing interface. The Config module is responsible for storing various configurations of the user. The Rules module is responsible for checking the rules of the code. The Proposal module is responsible for providing log statement suggestions. The Classifier module is responsible for training the recommended model. The Inspection module is responsible for scanning the code in real time and highlighting it. The Quickfix module is responsible for providing suggestions for modifications and automatically generating code. The Util module is responsible for providing shared logic.

4 PRELIMINARY EVALUATION RESULTS

In this section, we will show the actual application of the JLLAR tool in the open source project Tomcat.

After launching the JLLAR plug-in, it will inspect the code according to the rule set in real time. When existing violation code, it will show the yellow light bulb and provide quickfix including level recommendation to help generate logging statements. After optimizing rules with the machine learning technique, it can inspect the code more accurately.

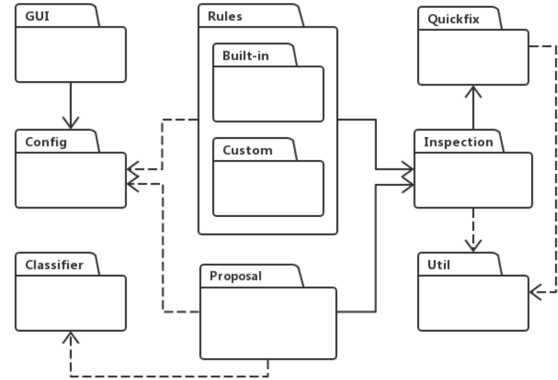

Figure 2: Architecture of JLLAR

4.1 Rule Inspection in the Catch Block

Taken this rule that exception needs to be logged as an example, as shown in Figure 3, it recommends to log the catch block because the developer did nothing before.

```
try {
    if (cl.getResource(path) == null) {
        return null;
    }
} catch (ClassCircularityError cce) {
}
```

It is recommended to log the catch block according to the rule set

Actually, developers did nothing in the catch block

Figure 3: Rule Inspection in the Catch Block

As shown in Figure 4, quickfix generates a logging statement with error level in the catch block that requires developer to fill some message.

```
try {
    if (cl.getResource(path) == null) {
        return null;
    }
} catch (ClassCircularityError cce) {
    LOGGER.error("Enter your message here!");
}
```

Quickfix automatically with inserting a logging statement

Figure 4: Quickfix in the Catch Block

After optimizing this rule, it is not recommended to log the catch block because in fact the developer handled the exception with a return statement, as shown in Figure 5.

```
try {
    clazz = cl.loadClass(name);
} catch (ClassNotFoundException e) {
    return null;
}
```

It is not recommended to log the catch block according to the rule set

Actually, developers handled the exception with return statement in the catch block

Figure 5: Rule Inspection in the Catch Block after Optimization

We count the number of issures before and after rule optimization in Tomcat. The result is shown in Table 5.

Table 5: Result of Rule Optimization in the Catch lock in Tomcat

Number of Issues Before Optimization	Number of Issues After Optimization
784	152

4.2 Rule Inspection in the If Block

Taken this rule that logic branch needs to be logged as an example, as shown in Figure 6, it recommends to log the if block because the developer didn't distinguish condition of the logic branch.

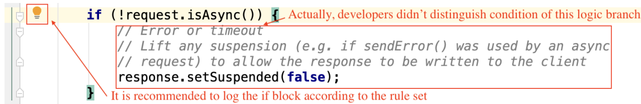

Figure 6: Rule Inspection in the If Block

As shown in Figure 7, quickfix generates a logging statement with error level in the if block that requires developer to fill some message.

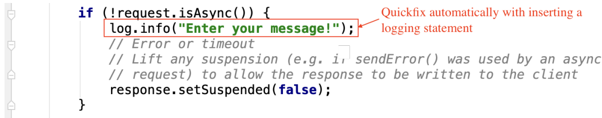

Figure 7: Quickfix in the If Block

After optimizing this rule, it is not recommended to log the if block because in fact the developer handled the logic branch with throw statement, as shown in Figure 8.

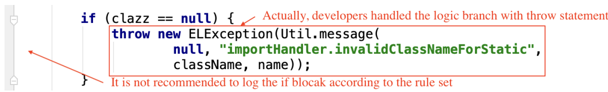

Figure 8: Result of Rule Optimization in the If Block in Tomcat

The result is shown of this rule's optimization in Table 6.

5 DISCUSSIONS

In this section, we will discuss the implication and threats to validity in our study.

5.1 Implication

The logging practice among distinct developers differ from each other to a very large extent. Not only the density of logging statements, but also the contents (what to log?) of logging statements vary among different developers[25]. With this ad-hoc manner to carry out the logging practice, we have to doubt whether logs can provide a reliable source of information for log analysis to establish a correct understanding of the runtime behavior of the program. In this study, we first and refine a rule set to help implementation of

Table 6: Optimization in Tomcat

Number of Issues Before Optimization	Number of Issues After Optimization
3847	261

several well recognized best practice on logging practice. Then we design a tool to help developers to do better logging in software development. Our work at this stage may reveal several interesting fact around logging practice in software development.

First, although the importance of logging practice is well recognized in software engineering, the implementation is far from satisfactory. In fact, when we evaluated *JLLAR*, we also identified many code snippets violating the logging rules we established in subsection 3.2.

Second, it seems that there is no simple solution to provide a "best" logging strategy. On the one hand, even in the aforementioned scenarios of rule violation, developers may choose not to put logging statements for the purposes of performance or security. On the other hand, the same rules may have some variations in different situations.

Last but not least, current logging practices often lack a comprehensive strategy, i.e., at present, we usually consider whether a logging statement should be inserted in a small code snippet and what the content of a logging statement. However, since there are multiple calling relationships between different levels of modules in one software system, it might not be an optimal choice to inject logging statements anywhere that meets the rules.

5.2 Threats to Validity

At this early stage, there are several considerations and limitations about both the research process and conclusions.

Project characteristics. Apparently, this study only involves 3 projects with Java language, which may have a limitation on generalization, i.e., to apply the results in other projects. However, since the projects involved in this study are very large in size, covering many logging scenarios, which may to a fair degree mitigate this limitation. As a matter of fact, the results of cross-project evaluation have shown that the model still has good applicability on different types of projects. Nevertheless, since the machine learning model is trained using Java language, we believe similar work on other languages should be conducted in future.

Open source projects. We carried out this study using open source projects, which to a certain degree limited our capability to generalize the results in this study to other scenarios, e.g., other open source projects and commercial projects. In particular, logging practice in commercial software project differ from open source projects greatly. In this sense, more investigation should be conducted with different types of software systems.

Field evaluation. At this stage, we only performed a preliminary evaluation by applying *JLLAR* in the projects involved in this study. However, we believe a gap may exist between the experimental evaluation and field adoption, i.e., to what degree, the developers will accept the recommended logging strategy. In fact, the revised rules are mainly based on a largely accepted logging strategy method.

Developers may have special considerations regarding the logging strategy. In this sense, it is necessary to carry out field evaluation on *JLLAR* as soon as possible.

6 CONCLUSION

Given the importance of the quality of logs and several issues existing in real-world logging practice, we devised a plug-in tool called *JLLAR* for the Java language on the Intellij IDEA platform, one of the most popular integrated development environment to support software developers to carry out logging practice. The main contributions of this paper can be highlighted as follows:

First, we extracted some best practices reported in relevant literature on logging practice and transformed these best practices into rules to guide logging practice.

Second, we applied machine learning technology to further refine the rules derived from recognized best practice regarding logging practice so that these rules could be adopted as similarly as possible to the actual placement of logging statements in open source projects.

Last but not least, based on the refined rule set, a plug-in tool was designed and implemented to support developers to carry out logging practice. For example, *JLLAR* can scan existing source code and identify missing logging statements or to insert/correct logging statement automatically.

There are still several limitations for this study at this preliminary stage, therefore, we suggest four major topics for future work.

- *JLLAR* need to be evaluated in real-world software development in order to collect developers' feedback and improve the tool.
- *JLLAR* has some built-in some logging rules, in order to solve some common problems, and finally achieve the objective of improving the quality of logging practice in software development. In the future, we still need to further refine the rule set to cover more coding scenarios.
- The accuracy of in the source project is about 95%, and the cross-project test also reaches 82%, but the shortcoming is that the accuracy of the logging statement level decision is only about 60%, which may be related to the choice of features, so it needs to be improved in the future.
- Since more and more programing languages are applied in modern software projects, *JLLAR* should be improved to support different programming languages, e.g., python, go, etc.

REFERENCES

[1] A. Ganapathi A. Oliner and W. Xu. 2012. Advances and challenges in log analysis. *Commun. ACM* 55, 2 (February 2012), 55–61.

[2] G. Carrozza A. Pecchia, M. Cinque and D. Cotroneo. 2015. Industry practices and event logging: Assessment of a critical software development process. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE '15)*. IEEE, 169–178.

[3] J. L. Hellerstein R. Rifaat B. Sharma, V. Chudnovsky and C. R. Das. 2011. Modeling and synthesizing task placement constraints in Google compute clusters. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing (SOCC '11)*. ACM, New York, NY, USA, 3:1–3:14.

[4] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (01 Oct 2001), 5–32. https://doi.org/10.1023/A:1010933404324

[5] B. Chen and Z. M. J. Jiang. 2017. Characterizing and detecting Anti patterns in the logging code. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE, Piscataway, NJ, USA, 71–81.

[6] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine Learning* 20, 3 (01 Sep 1995), 273–297. https://doi.org/10.1007/BF00994018

[7] P. Huang Y. Liu M. M. J. Lee X. Tang Y. Zhou D. Yuan, S. Park and S. Savage. 2012. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*. 293–306.

[8] S. Park D. Yuan and Y. Zhou. 2012. Characterizing logging practices in Open-Source software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 102–112.

[9] S. Park Y. Zhou D. Yuan, J. Zheng and S. Savage. 2012. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems* 30, 1 (1 February 2012), 4:1–4:28.

[10] Yoav Freund, Robert Schapire, and Naoki Abe. 1999. A short introduction to boosting. *Journal-Japanese Society For Artificial Intelligence* 14, 771-780 (1999), 1612.

[11] M. Goldszmidt J. Symons T. Kelly I. Cohen, S. Zhang and A. Fox. 2005. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. ACM, New York, NY, USA, 105–108.

[12] H. Idan. [n. d.]. Github research: Over 50% of java logging statements are written wrong. Retrieved February 7, 2017 from https://blog.takipi.com/github-research-over-50-of-java-logging-statements-are-written-wrong/

[13] Q. Fu H. Zhang M. R. Lyu J. Zhu, P. He and D. Zhang. 2015. Learning to log: Helping developers make informed logging decisions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE '15)*. IEEE, 415–425.

[14] C. Killian K. Nagaraj and J. Neville. 2012. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*. USENIX Association, Berkeley, CA, USA, 26–26.

[15] B. W. Kernighan and R. Pike. 1999. *The practice of programming*. Addison-Wesley Professional.

[16] J. L. Wiener P. Reynolds M. K. Aguilera, J. C. Mogul and A. Muthi tacharoen. 2003. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, New York, NY, USA, 74–89.

[17] D. Dagit R. B. Bobba M. Montanari, J. H. Huh and R. H. Camp bell. 2012. Evidence of log integrity in policy-based security monitoring. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN '12)*. IEEE, 1–6.

[18] Simon Maple. [n. d.]. Java Tools and Technologies Landscape Report 2016. Retrieved July 14, 2016 from https://jrebel.com/rebellabs/java-tools-and-technologies-landscape-2016/

[19] L. Mariani and F. Pastore. 2008. Automated identification of failure causes in system logs. In *2008 19th International Symposium on Software Reliability Engineering (ISSRE '08)*. IEEE, 117–126.

[20] Andrew McCallum, Kamal Nigam, et al. 1998. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, Vol. 752. Citeseer, 41–48.

[21] Andrew Y Ng and Michael I Jordan. 2002. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. In *Advances in neural information processing systems*. 841–848.

[22] A. J. Oliner and A. Aiken. 2011. Online detection of multi-component interactions in production systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*. IEEE, 49–60.

[23] W. Hu J.-G. Lou R. Ding Q. Lin D. Zhang Q. Fu, J. Zhu and T. Xie. 2014. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Confer- ence on Software Engineering (ICSE Companion 2014)*. ACM, New York, NY, USA, 24–33.

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

[25] Guoping Rong, Shenghui Gu, He Zhang, Dong Shao, and Wanggen Liu. 2018. How Is Logging Practice Implemented in Open Source Software Projects? A Preliminary Exploration. *2018 25th Australasian Software Engineering Conference (ASWEC)* (2018), 171–180.

[26] Guoping Rong, Qiuping Zhang, Xinbei Liu, and Shenghiu Gu. 2017. A Systematic Review of Logging Practice in Software Engineering. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 534–539.

[27] A. Fox D. Patterson W. Xu, L. Huang and M. I. Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 117–132.

[28] J. Zhan W. Zhou Z. Jia X. Fu, R. Ren and G. Lu. 2012. LogMaster:Mining event correlations in logs of Large-Scale cluster systems. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*. IEEE, 71–80.

[29] Y. Luo M. Stumm D. Yuan X. Zhao, K. Rodrigues and Y. Zhou. 2017. The game of twenty questions: Do you know where to log?. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*. ACM, New York, NY, USA, 125–131.