An Approach to Recommendation of Verbosity Log Levels Based on Logging Intention

Han Anu, Jie Chen, Wenchang Shi, Jianwei Hou, Bin Liang, Bo Qin
School of Information, Renmin University of China, Beijing, China
{ anuhan, jiechen1756, wenchang, houjianwei, liangb, bo.qin}@ruc.edu.cn

***Abstract*—Verbosity levels of logs are designed to discriminate highly diverse runtime events, which facilitates system failure identification through simple keyword search (e.g., *fatal*, *error*). Verbosity levels should be properly assigned to logging statements, as inappropriate verbosity levels would confuse users and cause a lot of redundant maintenance effort. However, to achieve such a goal is not an easy task due to the lack of practical specifications and guidelines towards verbosity log level usages. The existing research has built a classification model on log related quantitative metrics such as log density to improve logging level practice. Though such quantitative metrics can reveal logging characteristics, their contributions on logging level decision are limited, since valuable logging intention information buried in logging code context can not be captured.**

In this paper, we propose an automatic approach to help developers determine the appropriate verbosity log levels. More specially, our approach discriminates different verbosity log level usages based on code context features that contain underlying logging intention. To validate our approach, we implement a prototype tool, *VerbosityLevelDirector*, and perform a case study to measure its effectiveness on four well-known open source software projects. Evaluation results show that *VerbosityLevelDirector* achieves high performance on verbosity level discrimination and outperforms the baseline approaches on all those projects. Furthermore, through applying noise handling technique, our approach can detect previously unknown inappropriate verbosity level configurations in the code repository. We have reported 21 representative logging level errors with modification advice to issue tracking platforms of the examined software projects and received positive feedback from their developers. The above results confirm that our work can help developers make a better logging level decision in real-world engineering.

I. Introduction

Software logs are the main source information for a variety of analysis tasks, such as anomaly detection [1], failure diagnosis [2], [3], test analysis [4], [5] and performance issue identification [6], [7]. However, log messages are not of equal importance, as recording the highly diverse events which have varying degree severity. Therefore, to describe the different effects of logged states or actions, logging frameworks define verbosity levels for log messages [8]. For instance, when unexpected failures or potential problems occur in a system, logs should be assigned the less verbose levels (*fatal*, *error*, *warn*, etc.). On the contrary, for the purpose of tracking general information or debugging, the more verbose levels (*info*, *debug*, *trace*, etc.) are needed. The logging level design can assist operators in locating various runtime problems quickly and precisely when auditing highly diverse logs.

Generally, the verbosity level of logs is determined during the development of logging statements. The developers' decision on logging level assignments should be prudent, as a wrong verbosity level would cause a lot of redundant effort in maintaining software quality. The survey in [9] shows that the adjustments on verbosity level of logs accounting for one third of all log improvements. Furthermore, inaccurate log levels would confuse users and developers. For example, a Hadoop user complained that the system printed excessive logs with *error* levels even it performed in a secure way [10]. After a heated discussion on this issue, developers decided to lower the log level to anticipate possible misconceptions and ambiguities. Consequently, making the appropriate logging level decision is of vital importance.

However, to achieve such a goal is a challenging task for developers. There is currently no project-specific logging behavior instruction for them in the industry, even mature companies such as Microsoft [11]. The only reliable guidance is developers' experience and domain knowledge, which can not indicate the logging level decision properly, due to a lack of overall knowledge on the whole project. Besides, the suggestions given by the logging frameworks such as Log4j [12] and some online blogs [13], [14] are too high-level and abstractive to practice. For instance, the rule for *fatal* (or *error*) level has not helped developers to judge that under which degree of exception events they should "wake up" the administrator. Therefore, a practical strategy for logging level assignment has become an urgent demand for developers. In addition, we find that developers have a clear understanding on the purpose of the least (e.g., *fatal*) and the most verbose level (e.g., *debug* or *trace*). In contrast, they are frequently confused about the usage of the rest (e.g., *error*, *warn* and *info*) [14].

So far, few work has aimed at the improvement of the logging level practice. Yuan et al. proposed a simple checker [9] to detect inconsistent log level configurations in similar code snippets. Heng et al. [15] automatically suggested the appropriate verbosity log level for a newly-added logging statement through an ordinal regression model. Despite the innovative findings in these work, the logging level decision has not been adequately resolved. For instance, Verbosity Level Checker [9] can detect the problematic logging level practices, however, it does not provide the modification suggestion. The ordinal approach presented by Heng et al. is based on the log-related quantitative metrics such as log density. Though logging characteristics can be mined in the quantitative metrics, much

2576-3148/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSME.2019.00022

TABLE I
THE DETAILS OF STUDIED OBJECTS

Project	First Revision	SLOC	#Logging Statements	#Logging Statements Distribution				
				Exception Handling	Condition Check	Selection Control	Loop	Method
Hadoop	2006-07	1.1M	11370	3200 (28.14%)	4338 (38.15%)	131 (1.15%)	580 (5.10%)	3121 (27.45%)
Tomcat	2003-03	0.3M	2669	802 (30.05%)	1666 (62.42%)	24 (0.90%)	43 (1.61%)	134 (5.02%)
Qpid	2009-02	0.3M	2633	654 (24.84%)	1370 (52.03%)	3 (0.11%)	72 (2.73%)	534 (20.28%)
ApacheDS	2006-01	0.2M	2176	521 (23.94%)	1144 (52.57%)	20 (0.92%)	39 (1.79%)	452 (20.77%)
Total	-	1.9M	18848	5177 (27.48%)	8518 (45.19%)	178 (0.01%)	734 (0.04%)	4241 (22.50%)

valuable logging intention information that buried in logging code context can not be captured. Such code context is a significant factor of logging level decision according to our observations (in Section III-C). Besides, to the best of our knowledge, there exists no prior work studying the usage of ambiguous verbosity log levels (e.g., *error*, *warn* and *info*).

In this paper, we propose an automatic approach based on logging intention to assist developers in discriminating the usages of ambiguous log verbosity levels without requiring any domain knowledge. Especially, our approach extracts the contextual features from logging code snippets and leverages a machine learning model to automatically predict the verbosity level of logging statements.

As a proof of concept, we design and implement a logging level prediction tool, *VerbosityLevelDirector*, and validate its discrimination performance on four well-known open source software projects from Apache Software Foundation. Evaluation results show that *VerbosityLevelDirector* outperforms the baseline approaches on all studied projects, as we filter the irrelevant features and handle the noises in collected instances. Furthermore, we have reported the representative parts of logging level errors with modification advice to issue tracking platforms of the examined projects and received positive feedback from their developers. The above promising results demonstrate the discriminability and effectiveness of our approach.

The rest of the paper is organized as follows. Section II surveys related work. Section III describes the observations and motivation. Section IV presents our approach in detail. Section V discusses the experimental evaluation of *VerbosityLevelDirector* and SectionVI shows the threats to validity. Finally, we conclude this paper and discuss future work in Section VII.

II. RELATED WORK

Current log-related research can be roughly classified into three parts: (1) log analysis, (2) logging practices characterization and (3) logging practices improvement.

A. Log analysis

Console logs are used for problem identification of large scale systems or large scale data analytics applications in recent works. Xu et al. [1] utilized the features extracted from the console logs to detect an anomaly. Shang et al. [2] grouped log lines into execution sequences of pseudo and production environment, then identified the differences between two clusters to explain deployment failures. Yuan et al. designed SherLog, which could diagnose error through analyzing logs from failed production run [3]. Researchers have also studied logs for test analysis. Jiang et al. leveraged test logs to construct Cause Analysis Model [4] for root cause analysis. Anderson et al. built a classifier based on the attributes of test logs to predict test case failures [5]. Besides, logs are also useful for performance issues diagnosis [6], [7]. The above research has indicated that logs are widely studied in different fields of studies, thus maintain the quality of logging statements is of crucial importance.

B. Logging Practices Characterization

Logging practices are indispensable to the software development lifecycle, and understanding the real-world logging practices can facilitate good quality logging. The work [9] proposed by Yuan et al. is the first step towards characterizing the logging behavior by investigating and quantifying the logging changes in the code repository. They found that logging statements often experience the modification owing to the inappropriate decision in developers' first attempts. Chen and Jiang investigated the anti-patterns (e.g. nullable objects) in the independently changed logging code and designed a static code analysis tool, LCAnalyzer for detection [16]. Likewise, Kabinna et al. demonstrated the characteristics of changed logging statements and assisted developers of log processing tools in selecting more stable logs for analysis [17]. He et al. investigated the usage of static descriptions of logging content and applied a static model to capture the repetitive patterns [18].

Some work reveals the interaction of logging practices and other code-related resources [19]–[21]. Li et al. demonstrated that code snippet topics influence the logging decision [19]. Shang et al. [20] found that there exists a strong correlation between logging metrics and post release defects. They advised developers should give more maintaining attention to the files containing the logging statements. Shang et al. [21] provided a taxonomy on user inquiries of logs and associated the development knowledge with understanding the logging practices.

Except for the studies on web servers and desktop applications, Zeng et al. characterized the logging practices in Android applications [22]. They provided the performance evaluation on unnecessary logging and investigated the rationals of mobile logging.

C. Logging Practices Improvement

We break this part of work into the following three classes.

- *Logging Placement Study* focuses on the insertion point decision of logging statements. For instance, Errlog [23] added logging statements to the unlogged exception which matched their error patterns. Zhu et al. designed LogAdvisor [11] to identify the logging points by leveraging the contextual features from the source code. Smartlog [24] tackled this problem by mining rules from the equivalent logging intentions. Log20 [25] inserted non-erroneous logging statements to disambiguate the execution paths. Li et al. logged code snippets by applying the topic modeling approach [19].
- *Logging Content Study* focuses on the description texts and logged variables to improve existing logging statements. LogEnhancer [26] and Log20 [25] automatically added causality-related variable values to logging statements to ease system failure diagnosis. He et al. [18] proposed a simple information retrieval method based on their logging patterns findings to demonstrate the potential of automated logging description generation.
- *Logging Level Study* focuses on the verbosity log level examination and assignment. LCAnalyzer [16] checked the wrong verbosity level by whether the logging description and the level matched or not. Verbosity Level Checker [9] detected the inconsistent log levels in similar code snippets. Li et al. [15] collected the quantitative metrics (e.g, log density) and conducted a regression model to predict the verbosity level for newly added logging statements.

Our work is different from the aforementioned logging level studies. We make the logging level decision by understanding the logging intention. Furthermore, we can not only identify the problematic logging level practices, but also predict the appropriate verbosity log level for both problematic and new logging instances.

III. Observations and Motivation

A. Subject Projects and Logging Statements Distribution

In this paper, we study four popular open source software projects, namely Hadoop, Tomcat, Qpid and ApacheDS, each of which has a long development history (as shown in Table I). These projects cover different application domains, which can illustrate the generality of our work. In particular, Hadoop is a distributed computing framework for large data process. Tomcat is a lightweight web application server and a preferred tool to develop and debug JavaServer Pages. Qpid is a Message-Oriented Middleware (MOM) system, which supports many languages and platforms. ApacheDS is an embeddable and extendable modern Lightweight Directory Access Protocol (LDAP) server.

First, we investigate the logging practices in each software system to demonstrate the importance of logging. Table I presents the details of subject projects with the source lines of code (SLOC) and the count of the logging statements. We find that there is a positive correlation between the count of

```
/* Code Example 1: Exception Handling Block
 * hadoop-hdfs-project/.../BookKeeperJournalManager.java
 */
private void cleanupLedger(LedgerHandle lh) {
  try {
        long id = currentLedger.getId();
        currentLedger.close();
        bkc.deleteLedger(id);
      } catch (BKException bke) {
        // log & ignore, an IOException will be thrown
        LOG.error("Error closing ledger", bke);
      } catch (InterruptedException ie) {
        Thread.currentThread().interrupt();
        LOG.warn("Interrupted while closing ledger", ie);
      }
}
```

```
/* Code Example 2: Condition Checking Block
 * hadoop-common-project/.../StaticUserWebFilter.java
 */
static String getUsernameFromConf(Configuration conf) {
  String oldStyleUgi = conf.get(DEPRECATED_UGI_KEY);
  if (oldStyleUgi != null) {
    // We can't use the normal configuration ...
    // since we need to split out the username from ...
    LOG.warn(DEPRECATED_UGI_KEY +
    " should not be used. Instead, use " +
    HADOOP_HTTP_STATIC_USER + ".");
    String[] parts = oldStyleUgi.split(",");
    return parts[0];
  } else {
    ...
  }
}
```

Fig. 1. Focused Code Blocks with Logging Statement

the logging statements and the size of source code. It indicates the larger the software system is, the more logging practices are accomplished to record operation behaviors of the system. Then, we analyze the pervasiveness of logging statements in different code blocks. As shown in Table I, the logging statements are scattered across every code blocks. Note that, the great majority of them are distributed in the *Exception Handling block* and the *Condition Check block* in each project, accounting for 66.36% to 92.47%. The main reason of such distribution is the functionality of these two blocks is designed to capture unexpected exceptions or abnormal return values during the runtime. The logging statements are inserted in these two blocks in order to collect the worthy context information for postmortem root cause analysis. Thus, to make our logging level study general to most of the logging practices, we focus on the *Exception Handling block* and the *Condition Check block* of the subject projects.

B. Verbosity Log Level Distribution in Focused Code Blocks

All of the studied projects use the standard and prevalent logging libraries to implement logging practice, such as Apache Log4j [12], SLF4J [27] and Apache Commons Logging library. These logging libraries define six verbosity log levels for logging statements, from *trace* to *fatal*. In Table II, we investigate the distribution of the six verbosity levels in the focused code blocks (mentioned in Part A). The verbosity

TABLE II
VERBOSITY LOG LEVEL DISTRIBUTION IN FOCUSED BLOCKS IN THE STUDIED PROJECTS

Verbosity Level	Hadoop	
	Exception Handling	Condition Check
fatal	61 (1.91%)	26 (0.60%)
error	837 (26.16%)	352 (8.11%)
warn	804 (25.13%)	877 (20.22%)
info	1182 (36.94%)	1326 (30.57%)
debug	268 (8.38%)	1594 (36.75%)
trace	48 (1.50%)	163 (3.76%)

Verbosity Level	Tomcat	
	Exception Handling	Condition Check
fatal	5 (0.61%)	4 (0.24%)
error	494 (60.10%)	164 (9.84%)
warn	227 (27.62%)	264 (15.85%)
info	32 (3.89%)	162 (9.72%)
debug	42 (5.11%)	957 (57.44%)
trace	22 (2.68%)	115 (6.90%)

Verbosity Level	Qpid	
	Exception Handling	Condition Check
fatal	0 (0.00%)	0 (0.00%)
error	247 (37.77%)	65 (4.74%)
warn	120 (18.35%)	96 (7.01%)
info	91 (13.91%)	169 (12.34%)
debug	187 (28.59%)	484 (35.33%)
trace	9 (1.38%)	556 (40.58%)

Verbosity Level	ApacheDS	
	Exception Handling	Condition Check
fatal	0 (0.00%)	0 (0.00%)
error	281 (53.93%)	239 (20.89%)
warn	92 (17.66%)	110 (9.62%)
info	40 (7.68%)	106 (9.27%)
debug	103 (19.77%)	684 (59.79%)
trace	5 (0.96%)	5 (0.44%)

levels with less frequency usage are *fatal* (0.00% to 0.24%) and *trace* level (0.44% to 40.58%). The former is used only when the application or system has failed in a way that should be terminated and investigated immediately [14]. As for *trace* level, it is used for tracing purposes that are determined by the needs of developers. Though *debug* level appears at high frequency in *Condition Check block* of all projects, through our manual analysis, we find that most of the instances check whether the current logger is enabled for *debug* level or not (e.g., *isDebugEnabled()*). In other words, such *Condition Check block* instances are under the logging framework mechanism, which is independent of the logging decision.

The rest of the levels, *error*, *warn* and *info*, of gradually decreasing severity, are widely used in the focused code blocks. According to the discussion on StackOverflow [14], we observe that developers have varying opinions on the usages of these three verbosity levels. It indicates that the usage of the specific verbosity levels is still an ambiguous decision in real-world engineering. Consequently, driven by the above findings and the real-world issues, we focus on the decision of *error*, *warn* and *info* level in this paper.

C. The Logging Context and Logging Level Decision

To understand logging level decision in the focused code blocks, we analyze the existing logging practices of studied projects, the logging quality of which is ensured by a lot of experienced developers. Fig. 1 shows an *Exception Handling block* and a *Condition Check block* examples from the real-world instances. Both of them leverage the logging statement to print the failure related message after capturing an exception or checking an unusual value. As Code Example 1 presents, the logging statement in the first catch block logged a *BKException* event. The operations trigger this exception handling mechanism can be found in the *try* block. Clearly, the called method *close()* would be the preferred candidate compared with *getId()* and *deleteLedger()*. Since its function name and the static description text of the logging statement are semantically equivalent. Notably, though the logging statements in distinct catch blocks (capture the *BKException* and *InterruptedException*, respectively) share the similar logging content, their verbosity levels are quite different. This example fully illustrates the relationship between logging practices and logging code context (triggered methods, exception type, etc.), which inspires our work to learn logging level decision based on such code contexts.

D. Motivation

In spite of the vital importance of logging level decision, there are no specifications for developers during the lifecycle of software development. New practitioners learn the domain knowledge and experiences only from some skilled developers. The top answers to logging level decision question on StackOverflow [13] explained that if a system failure needs to "wake up" the developers, then assign a less verbose level to logging statement. This suggestion gives a lively description of the verbosity level application. However, without more details, it is still tough to guide the actual practices, especially for new developers. Besides, from the discussion under similar questions, we notice that the specific verbosity level, such as *error*, *warn* and *info* are more ambiguous to the developers. Motivated by this situation, we learn the existing logging statements and their associated code contexts in our approach to help developers make the appropriate logging level decisions without requiring any priori knowledge. During the research, we find that although the majority of practices are designed well, there remain some improper designs. For example, some verbosity log levels of logging statements are inconsistent but share a similar context. The inconsistent logging levels would cause confusion in the future, which should be identified and modified earlier. Therefore, to maintain the good logging level practices, our approach also checks the problematic logging levels in the studied software and provides modification suggestions.

IV. OUR APPROACH

In this section, we present the overview and detailed techniques of our approach.

A. Overview

Fig. 2 illustrates an overview of our logging level decision approach in the following steps. First, we recognize the focused

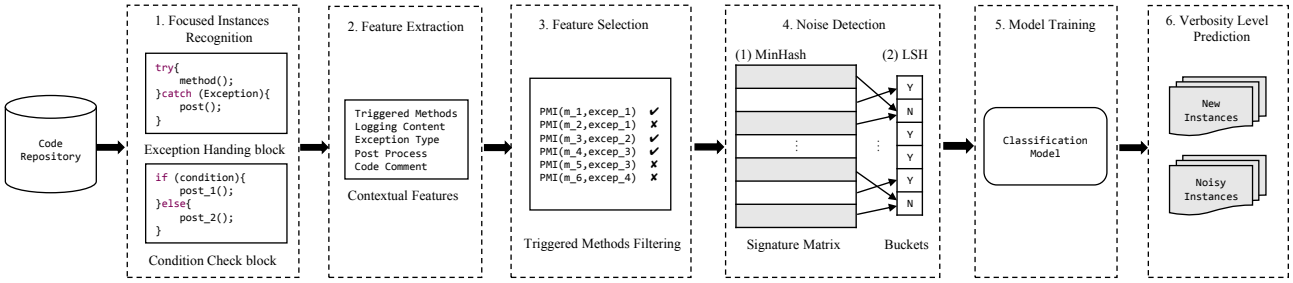

Fig. 2. The Approach Overview

code blocks in subject projects. Second, we extract contextual features which imply the logging intention. Third, we employ a feature selection algorithm to filter the features which irrelevant to logging level decision. After that, we apply the noise handling technique to detect inappropriate verbosity level configurations and then train a classification model. Finally, we apply the model on both test and noise instances to predict their verbosity log levels. Additionally, we report logging level errors with modification advice to issue tracking platforms of the examined projects, such as JIRA [28] and Bugzilla [29].

B. Focused Instances Recognition

To understand the logging level decision in real-world projects, the first step is to recognize the focused code blocks: *Exception Handling blocks* and *Condition Check blocks*. We leverage a static code analysis technology, Abstract Syntax Tree (AST) [30], to identify all data instances from the code repository. During this step, we find that logged instances are a minority in the collected focused blocks. This causes the data imbalance issue that would affect the subsequent processes. Therefore, to alleviate the data defect, we utilize unlogged instances to get more code context information for logging level decision.

C. Contextual Feature Extraction

Logging level decision is closely related to logging intention which is generally buried in logging code context. To learn such intention, we extract the following features from the code contexts for model training:

1) ***Triggered Methods***

The logging code context is mainly composed of the called methods in the focused code blocks. Some of these methods return abnormal values or trigger exception handling mechanisms, are referred as the triggered methods, i.e., checked condition of *if (else)* block and called methods in *try* block. The triggered methods are the decisive factor of logging intention, as their various states have to recorded in logging statements. Thus, we extract them as significant contextual features.

Notably, when we extract the triggered method of *if (else)* block, we find that it can not be directly obtained in some cases, since the checked condition is a variable not a method. Thus, we use program dependence analysis technique to scan code context forwards from the condition variable, identifying the triggered methods with flow dependence. Code Example 2 in Fig. 1 presents a logged instance of *Condition Check block*. The triggered method in this block is the called method *get()*, because it assigns the function return value to the condition variable *oldStyleUgi*.

To get more detailed information about the triggered methods (e.g., defined package name and class name), we obtain their full qualified names which also can distinguish methods with the same name.

2) ***Logging Content***

The logging content is the important contextual information for developers and users to diagnose a fault [8]. It is composed of static text and variable values. As the logging statement shown in Code Example 2, the logged variable value has input dependency on the triggered method, which can help developers to locate failure causes in postmortem analysis. For another logging content, the static text, giving a semantic description for methods (e.g., *Interrupted while closing ledge*). Besides, the static text also indicates the severity of exception sometimes (e.g., *error*, *terrible*, etc.) Though verbosity log level can not be directly determined by these static texts, they are non-ignorable elements to logging level decision. We hence extract the logging content of all logged instances as one of the contextual features.

3) ***Exception Type***

We also extract the exception type of the *Exception Handling block*, which presents what exception is captured by the catch clause. Sometimes, called methods in *try* block might trigger more than one type of exceptions. Thus to handle the variety of possibilities, the *try* block usually needs several catch clauses with different exception types. For instance, in Code Example 1 in Fig. 1, two different catch clauses capturing *BKException* and *InterruptedException*, respectively. Both of them have a logging statement with different verbosity levels. It means that even though catch clauses share the identical triggered methods, the verbosity log level assignment is still yet to be decided due to the exception type.

4) ***Post Process***

The discussions on StackOverflow [13], [14] suggest that, if an execution failure would affect user operations, the logging statement should be assigned to *error* level, otherwise *warn* level is needed. Thus, to understand the failure affection on logging level decision, we try to give a first step study in

```
/* Code Example 3: involving lots of called methods
 * hadoop-mapreduce-project/.../ClientServiceDelegate.java
 */
private MRClientProtocol getProxy() throws IOException {
  ...
    try {
      if (application.getHost() == null ||
          "".equals(application.getHost())) {
        Thread.sleep(2000);
      }
      if(!conf.getBoolean(MRJobConfig.
      JOB_AM_ACCESS_DISABLED, false)) {
        UserGroupInformation newUgi = UserGroupInformation
        .createRemoteUser(
        UserGroupInformation.getCurrentUser()
        .getUserName());
        serviceAddr = NetUtils.createSocketAddrForHost((),
        application.getRpcPort());
        if (UserGroupInformapplication.getHostation
        .isSecurityEnabled()) {
          newUgi.addToken(token);
        }
      }
    } catch (InterruptedException e) {
      LOG.warn("getProxy() call interruped", e);
      throw new YarnRuntimeException(e);
      }
      ...
  return realProxy;
}
```

(a) Exception Handling Block with Lots of Triggered Methods

```
Positive Strong Correlation:
(sleep(), InterruptedException): 3.41
(getHost(), InterruptedException): 3.32
(getRpcPort(), InterruptedException): 3.32
(createSocketAddrForHost(), InterruptedException): 2.03

Positive Weak Correlation:
(createRemoteUser(), InterruptedException): 0.89
(isSecurityEnabled(), InterruptedException): 0.77
(addToken(), InterruptedException): 0.23
(getBoolean(), InterruptedException): 0.15
(getUserName(), InterruptedException): 0.07

Negative Correlation:
(getCurrentUser(), InterruptedException): -0.24
(equals(), InterruptedException): -1.37
```

(b) PMI Correlation Between Triggered Method and Exception

Fig. 3. Example of Triggered Method Selection

this paper. We observe that the contexts in *if (else)* block and *catch* clause reveal parts of the post effects of the unexpected error, which is referred as post process in our approach. A post process might include called methods and return statement (shown in the *if* block of Fig. 1 Code Example 2) or throw statement (shown in the *catch* clause of Fig. 3 Code Example 3).

*5) **Code Comment***

In addition, we extract code comments of focused code blocks as an extra contextual feature. Unlike the source code, the comments give a natural interpretation on the intention of logging. Therefore, code comments can provide more semantic rich information to logging level decision.

D. Feature Selection

After extracting contextual features, each logged instance is represented by them. However, some of the features are not relevant to the logging level decision, which should be gotten rid of to improve the performance of the classification model. These irrelevant features exist in the triggered methods of *Exception Handling block*. It is because some called methods need to present as the necessary auxiliary operations of a function but have little correlation with the caught exceptions. Our approach encounters a challenge when identifying such irrelevant methods. Since except for logging statements, there are no useful clues for discovering the correlation between triggered methods and caught exception. However, even the logging statement does not help sometimes. Fig. 3 (a) gives an example of *Exception Handling block* with lots of called methods in the *try* block. As we can observe, the logging statement in the *catch* block does not present extra information about the triggered methods, which makes it more difficult to find out the irrelevant calls.

To address this challenge, we need to identify the called methods which have a strong correlation with the caught exception. In our approach, we employ Pointwise Mutual Information (PMI) metric to reveal this correlation. PMI is used for finding word association in computational linguistics originally. It can take positive or negative values, and is zero if two words are independent. Word pairs with strong correlations have high positive PMI. Conversely, a pair of words have negative PMI if mutually excluded. In this paper, we use PMI to measure the association between the called methods in *try* block and a certain type of exception. More practically, we reserve the calls with high PMI values in the feature set, and filter those with PMI values under the threshold. As the example of PMI metric application shown in Fig. 3, the called methods with high PMI values such as *sleep()*, *getHost()*, *getRpcPort()* and *createSocketAddrForHost()* are more likely to actually trigger the exception handler, while those such as *getUserName()* are approximately independent of the current exception, which would be dropped from the feature set.

Furthermore, PMI value can assist developers in quickly locating the triggered methods. In Fig. 3(a), when the *Exception Handling block* catches an *InterruptedException* error, the logging statement records the exception message with a *warn* verbosity level. However, it only records the focused block containing method *getProxy()* without more details about the triggered methods in the *try* block. By comparison, according to the top ranked PMI values in the first listing of Fig. 3 (b), developers can find out the failure position rapidly and precisely.

Besides filtering the triggered methods, some worthless features (e.g., measure words and conjunction words) involved in the code comments and static logging text should be removed from the final features. The recent research such as [11] processes source code features as flat text. Motivated by this, we convert the contextual features to textual features and then filter them by applying a widely used text processing technology

named bag-of-words model.

E. Noise Detection and Modification

The next challenge we are facing is noise data detection. Since the origin data set is not perfect due to various factors, such as existing programming errors of data. The imperfect part of them called noises, which could decrease model reliability. Therefore, noises should be handled properly. Yuan et al. [9] found that there are inconsistent logging level practices in similar source code snippets. Driven by this observation, we consider such problematic instances as noises, because they could cause negative impact on classification performance. Hence such instances should be detected before the training process.

To identify the noises, the primary task is to find similar code blocks. To achieve this, we employ clustering in our study. Clustering is a common technology for data mining. It groups data objects into multiple clusters, and the objects in the same cluster have higher similarity than to those in others. Since our data are high-dimensional, composed of more than a thousand features, the traditional clustering algorithms are not applicable to our case. Therefore, we adopt high-dimensional data clustering techniques, MinHash and Locality-sensitive hashing (LSH), to resolve this issue. First, we use MinHash to reduce the dimension of features. The whole data set would be represented as a MinHash signature matrix, the dimension of which is far lower than the initial. Subsequently, LSH hashes the signature matrix to map the similar sets into the same buckets that the clusters with similar code blocks. After obtaining these clusters, we check whether the verbosity levels of instances in the same bucket are consistent or not. If not, all of the instances in the bucket are labeled to noise data.

Unlike other research, such as *learning to log* [11], the noise data are not directly dropped in our approach. Since these data only need to be adjusted in a few places (e.g., verbosity log level), removing all of them would miss much valuable information. To reuse the noises in our approach, we predict more appropriate verbosity levels for them through applying our trained classification model. Finally, we report these previously unknown logging level errors with modification advice to issue tracking platforms of the examined software.

In addition, our noise handling technique allows more insignificant differences in similar instances than Verbosity Level Checker [9]. The Verbosity Level Checker only detects the copy-pasted snippets from the source code and limits the difference to one statement. However, some snippets have broadly the same context when excluding the irrelevant called methods. Therefore, to identify more similar snippets, we set the detection granularity to the triggered methods which are preprocessed by the feature selection step.

V. Evaluation

A. Approach Performance Measurement

1) Experiment Setup

Focused instance identification and feature extraction are completed by AST [30] and SOOT [31] which are well-known static java code analysis tools. AST could help us to locate the *Exception Handling block* as well as the *Condition Check block*. SOOT performs feature extraction and data-flow analysis. To make better use of SOOT in our study, we compile the source codes of the studied projects to Jimple intermediate representation before employing it. After the feature extraction and noise detection, we can obtain the input set with higher quality for model building.

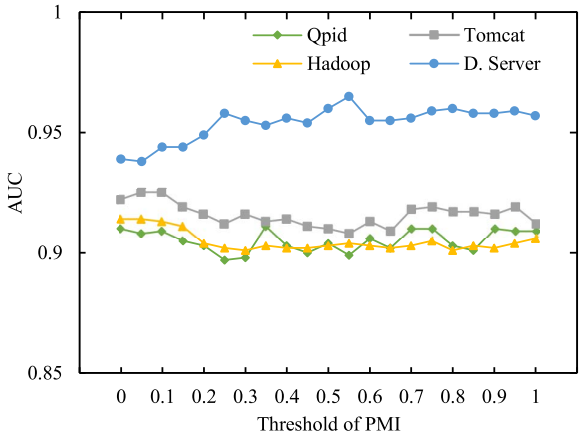

Fig. 4. Threshold Impact on Classification Effect

In this paper, the logging level decision is treated as a multi-classification task. More specifically, we build a Random Forest model to predict the verbosity level of given logging instances, due to its good classification effect. Since real-world sample distribution is not perfect, there is a class imbalance problem in our collected instances. For example, the instances logged with *error* and *warn* levels are much more than those with *info* level in Tomcat project. In such case, the normal metrics (e.g., *accuracy*) can not reflect model performance. Thus, we use AUC (area under the ROC curve) to evaluate the classifier, which is insensitive to changes in class distribution [32]. The value of AUC ranges from 0 to 1, the closer to 1 the value is, the more well-discriminated the model is. Since logging level decision is a multi-classification problem, we adopt the measure proposed by Hand et al. [33] to calculate the multi-class AUC value:

$$AUC_{multi} = \frac{2}{c(c-1)} \sum AUC_{i,j} \tag{1}$$

where c is the number of classes, $AUC_{i,j}$ is the area under the ROC curve involving class pair (i, j). There are $c(c-1)/2$ such pairs and AUC_{multi} is the average of their area summation.

2) PMI Threshold Assignment

In our approach, only the threshold of PMI needs to be determined. According to the PMI definition presented in Section IV-B, we first remove the triggered methods with negative PMI value. Then the PMI values of the rest have to be normalized, as they are typically out of the range of 0 to 1. In particular, we use min-max scaling for the

normalization process and run the model under different PMI values with 0.05 step length. As Fig. 4 shows, the horizontal axis indicates the association degree between the triggered method and the specific exception type. The vertical axis presents classification performance on different projects. When choosing the optimized PMI threshold, except for the classification performance, the filtering ability of our approach also should be taken into account. Because filtering irrelevant triggered methods could shorten troubleshooting time for developers. Therefore, under the prerequisite of high AUC guaranteed, we set the PMI threshold to 0.75 to retain only those triggered methods which have a strong correlation with the exception type.

*3) **Comparison Experiments***

As a proof of concept, we implement a prototype tool, *VerbosityLevelDirector*, for verbosity level prediction. We compare *VerbosityLevelDirector* with two baseline approaches: a random guess model and a raw model. As for the random guess model, we imitate the real-world condition that a developer has no prior knowledge on verbosity log level usages and predicts the verbosity level for a logging statement randomly. In the raw model, we cut off feature selection and noise handling step from our approach, in other words, building the model on raw data. All experiments are evaluated by 10-fold cross-validation mechanism [34] and the results are shown in Table III.

For all the examined projects, *VerbosityLevelDirector* outperforms the baseline approaches, exceeding 0.47 on AUC value at most. The raw model performs better than the random guess model. However, it does not reach the highest AUC value, because it disregards the necessary steps, i.e., the feature selection and noise detection. The experimental results indicate that our approach can help developers make appropriate decisions on the verbosity level assignment.

To further confirm the effectiveness of the selected classification model in our study, we also build several popular learning models on the same data sets to estimate their performance. The examined models include Naive Bayes, Decision Tree, Logistic Regression and Support Vector Machine(SVM) model.

TABLE III
MULTI-CLASS AUC VALUE OF DIFFERENT APPROACHES

	Hadoop	Tomcat	Qpid	ApacheDS
Random guess	0.50	0.50	0.50	0.50
Raw model	0.90	0.91	0.90	0.94
VerbosityLevelDirector	**0.90**	**0.92**	**0.92**	**0.97**

TABLE IV
MULTI-CLASS AUC VALUE OF DIFFERENT MODELS

	Hadoop	Tomcat	Qpid	ApacheDS
Naive Bayes	0.67	0.73	0.81	0.76
Logistic Regression	0.79	0.71	0.70	0.82
Decision Tree	0.78	0.81	0.79	0.86
SVM model	0.76	0.81	0.83	0.87
Radom Forest	**0.90**	**0.92**	**0.92**	**0.97**

TABLE V
REPORTS AND FEEDBACK

	Hadoop	Tomcat	Qpid	ApacheDS	**Total**
Reports	6	10	4	1	21
Fixed	6	10	0	1	17
Under Review	0	0	4	0	4
New Patches	0	8	0	0	8

TABLE VI
REPORTS INVOLVING DIFFERENT VERBOSITY LEVELS

	Hadoop	Tomcat	Qpid	ApacheDS	**Total**
(info, warn)	2	0	0	0	2
(info, error)	0	1	1	0	2
(warn, error)	4	9	3	1	17

As the results presented in Table IV, all models have good discriminability. Nevertheless, Naive Bayes and Decision Tree do not match with our instances, because both of them dismiss the relation among features. Due to the fact that Logistic regression is a generalized linear model, it does not work better on our nonlinear structural data. Although SVM is able to cope with the high-dimensional features, its results are difficult to interpret by comparing with Random Forest model. Above all, the Random Forest became the optimal choice in our case.

B. Noise Handling Effects

Through applying the noise handling technique, our approach can detect previously unknown inappropriate logging level practices. We apply the trained model on them to predict a more appropriate verbosity level. As Table V shows, we have reported 21 representative errors with modification advice to issue tracking platforms of the studied projects. Except for the errors reported to Qpid project, others have been already fixed by the developers of the corresponding projects, and 14 of the modification advice have been adopted in fixed revisions. In addition, inspired by our findings, developers have detected 8 new logging level errors and add patches for them.

We further break down the reports between any two verbosity levels in Table VI. Most (80%) of the problematic practices are between *warn* and *error*. This pair is different from others in that the severity of the runtime event is hard to assess during the development lifecycle. The uncertainty is reconsidered by developers only when having difficulty in failure diagnosis [9]. Therefore, our noise handling approach is of great importance, as it detects and modifies inappropriate verbosity level assignment before various system analysis. Besides, our results advocate the need of more rigorous guidelines for ambiguous verbosity levels usages, especially *warn* and *error*.

VI. THREATS TO VALIDITY

Logging Quality of Studied Projects: Owing to the lack of standard documents for logging level decision, the only available resource to our study is the logging practices in the real-world projects. We assume that the examined projects

have high-quality logging practices because of their mature design and well maintenance. Though there exist inappropriate practices, we can ease this issue by noise handling technique. The encouraging evaluation results and positive developer feedback also can prove logging code quality of the studied projects.

Generality of Approach: We select the studied open source projects from different domains, such as distributed computing, web application server, middleware technology, and directory server. These projects are regarded as the representatives in their domains, which ensures that our work can be applied in a wide range of software, not limited to the specific ones. We need to point out that the subject projects are all written in Java language, which might make our study look not general enough. However, we believe that the core concept of our approach can be ported to the projects developed by other programming languages. Since logging level decision is inseparable from its associated code context everywhere. In addition, our approach does not need to adjust the learning model parameters for the specific software. It can directly train the classification model on the filtered features and then assist developers in predicting verbosity levels for new logging instances.

Code Similarity vs. Function Similarity: Despite the effectiveness of noise handling in finding problematic logging level practices, it still has potential improvements that need further study and exploration. We only focus on code similarity at present when clustering the logging instances. The developers of the studied projects suggested that our work should give more attention to function similarity [35]. However, function similarity is more difficult to be characterized than code similarity. It needs a high-level understanding on code snippets. For instance, an invalid charset might be a minor issue when provided as part of a user request but a significant problem if a security realm is configured to digest passwords using an invalid encoding. Therefore, function similarity would be considered in our future work.

VII. Conclusion and Future Work

The verbosity log levels are of critical importance on highly diverse runtime event discrimination. However, software industries have no relevant specifications toward verbosity level assignments. In addition, logging level practices have not been properly resolved in current research. In this paper, we propose an automatic approach to make the appropriate logging level decision based on the associated code context which contains the logging intention of developers. As a proof of concept, we implement the prototype tool, *VerbosityLevelDirector*, to give informative guidance on verbosity log level assignments in focused code blocks. We evaluate its classification performance on four well-known software projects. The evaluation results and positive feedback from the developers of the examined projects can confirm the discriminability and effectiveness of our approach in real-world engineering.

In the future, we want to gather more feedback from developers to obtain insightful opinions for our work. Then, we plan to extract other valuable context features (e.g, function features and execution paths) to improve our approach. Finally, we also want to summarize the knowledge of logging level usage scenarios into a general reference for developers.

Acknowledgment

The authors thank the anonymous reviewers for their constructive views and suggestion. The work described in this paper was substantially supported by the National Natural Science Foundation of China (Project No. 61472429 and U1836209).

References

[1] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, 2009, pp. 117–132.

[2] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin, "Assisting developers of big data analytics applications when deploying on hadoop clouds," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 402–411.

[3] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: Error diagnosis by connecting clues from run-time logs," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 143–154.

[4] H. Jiang, X. Li, Z. Yang, and J. Xuan, "What causes my test alarm?: automatic cause analysis for test alarms in system and integration testing," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 712–723.

[5] J. Anderson, S. Salem, and H. Do, "Striving for failure: An industrial case study about test failure prediction," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, 2015, pp. 49–58.

[6] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. USENIX Association, 2012, pp. 26–26.

[7] M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. N. Nasser, and P. Flora, "Leveraging performance counters and execution logs to diagnose memory-related performance issues," in *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, 2013, pp. 110–119.

[8] "The art of logging," https://www.codeproject.com/Articles/42354/The-Art-of-Logging.

[9] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *2012 34th International Conference on Software Engineering (ICSE)*, June 2012, pp. 102–112.

[10] "Usergroupinformation prints out excessive error warnings," https://issues.apache.org/jira/browse/HADOOP-10015.

[11] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 415–425.

[12] "Apache log4j," http://logging.apache.org/log4j/1.2/.

[13] "Logging levels - logback - rule-of-thumb to assign log levels," https://stackoverflow.com/questions/7839565/logging-levels-logback-rule-of-thumb-to-assign-log-levels.

[14] "When to use the different log levels," https://stackoverflow.com/questions/2031163/when-to-use-the-different-log-levels.

[15] H. Li, W. Shang, and A. E. Hassan, "Which log level should developers choose for a new logging statement?" *Empirical Software Engineering*, vol. 22, no. 4, pp. 1684–1716, Aug 2017.

[16] B. Chen and Z. M. Jiang, "Characterizing and detecting anti-patterns in the logging code," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017, pp. 71–81.

[17] S. Kabinna, W. Shang, C. Bezemer, and A. E. Hassan, "Examining the stability of logging statements," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 326–337.

[18] P. He, Z. Chen, S. He, and M. R. Lyu, "Characterizing the natural language descriptions in software logging statements," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018, New York, NY, USA, 2018, pp. 178–189.
[19] H. Li, T. P. Chen, W. Shang, and A. E. Hassan, "Studying software logging using topic models," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2655–2694, 2018.
[20] W. Shang, M. Nagappan, and A. E. Hassan, "Studying the relationship between logging characteristics and the code quality of platform software," *Empirical Software Engineering*, vol. 20, no. 1, pp. 1–27, 2015.
[21] W. Shang, M. Nagappan, A. E. Hassan, and Z. M. Jiang, "Understanding log lines using development knowledge," in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sep. 2014, pp. 21–30.
[22] Y. Zeng, J. Chen, W. Shang, and T.-H. P. Chen, "Studying the characteristics of logging practices in mobile apps: a case study on f-droid," *Empirical Software Engineering*, Feb 2019.
[23] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, "Be conservative: Enhancing failure diagnosis with proactive logging," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, 2012, pp. 293–306.
[24] Z. Jia, S. Li, X. Liu, X. Liao, and Y. Liu, "SMARTLOG: place error log statement by deep understanding of log intention," in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, 2018, pp. 61–71.
[25] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "Log20: Fully automated optimal placement of log printing statements under specified overhead threshold," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, 2017, pp. 565–581.
[26] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, 2011, pp. 3–14.
[27] "Slf4j," https://www.slf4j.org.
[28] "Jira software," https://www.atlassian.com/software/jira.
[29] "Bugzilla," https://www.bugzilla.org.
[30] "Abstract syntax tree," https://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html.
[31] "What is soot?" https://sable.github.io/soot/.
[32] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006.
[33] D. J. Hand and R. J. Till, "A simple generalisation of the area under the ROC curve for multiple class classification problems," *Machine Learning*, vol. 45, no. 2, pp. 171–186, 2001.
[34] I. H. Witten, E. Frank, and M. A. Hall, *Data mining: practical machine learning tools and techniques, 3rd Edition*. Morgan Kaufmann, Elsevier, 2011.
[35] "Inconsistent log level practices in catalina component," https://bz.apache.org/bugzilla/show_bug.cgi?id=63287.