# DeepLV: Suggesting Log Levels Using Ordinal Based Neural Networks

Zhenhao Li*, Heng Li†, Tse-Hsun (Peter) Chen*, and Weiyi Shang*
*Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada
{l_zhenha, peterc, shang}@encs.concordia.ca
†Department of Computer Engineering and Software Engineering, Polytechnique Montréal, Montreal, Canada
heng.li@polymtl.ca

***Abstract*—Developers write logging statements to generate logs that provide valuable runtime information for debugging and maintenance of software systems. Log level is an important component of a logging statement, which enables developers to control the information to be generated at system runtime. However, due to the complexity of software systems and their runtime behaviors, deciding a proper log level for a logging statement is a challenging task. For example, choosing a higher level (e.g., *error*) for a trivial event may confuse end users and increase system maintenance overhead, while choosing a lower level (e.g., *trace*) for a critical event may prevent the important execution information to be conveyed opportunely. In this paper, we tackle the challenge by first conducting a preliminary manual study on the characteristics of log levels. We find that the syntactic context of the logging statement and the message to be logged might be related to the decision of log levels, and log levels that are further apart in order (e.g., *trace* and *error*) tend to have more differences in their characteristics. Based on this, we then propose a deep-learning based approach that can leverage the ordinal nature of log levels to make suggestions on choosing log levels, by using the syntactic context and message features of the logging statements extracted from the source code. Through an evaluation on nine large-scale open source projects, we find that: 1) our approach outperforms the state-of-the-art baseline approaches; 2) we can further improve the performance of our approach by enlarging the training data obtained from other systems; 3) our approach also achieves promising results on cross-system suggestions that are even better than the baseline approaches on within-system suggestions. Our study highlights the potentials in suggesting log levels to help developers make informed logging decisions.**

***Index Terms*—logs, deep learning, log level, empirical study**

## I. Introduction

Software logs have been widely used in practice for various maintenance activities, such as testing [1]–[5], failure diagnosis [6]–[9], and program comprehension [10], [11]. Developers insert logging statements in the source code with different verbosity levels (e.g., *trace*, *debug*, *info*, *warn*, *error*, and *fatal*) to record system execution information and values of dynamic variables. For example, in the logging statement: *LOG.info("stopping server ", + serverName)*, the static text message is *"stopping server "*, and the dynamic message is the value of the variable *serverName*. The logging statement is at the *info* level, which is the level for recording informational messages that highlight the progress of the application at a coarse-grained level [12].

Log levels enable developers to only print important log messages (e.g., error or warning information) at runtime while suppressing less important messages (e.g., debug messages). It is important for developers to choose the right log levels for their logging statements. On one hand, choosing a lower log level (e.g., *debug*) for a critical event can hide important runtime information and make it difficult to diagnose runtime failures [8]. On the other hand, choosing a higher level (e.g., *warn*) for a trivial event can confuse end users and increase the overhead of log management and analysis [13].

However, it is usually challenging for developers to choose a proper log level for the logging statements [13]–[16]. Prior studies shows that developers may not have sufficient understanding of the runtime behaviors of their systems and the purposes of different log levels [13], [15], leading to suboptimial choices of log levels. In particular, prior work [13], [14], [16] observes that developers spend significant efforts in modifying the levels of existing logging statements, as it is challenging for them to make the right decisions in the first place.

In this paper, we conduct a study to help developers make informed decisions on deciding proper log levels. Through a preliminary manual study on the logging statements from nine open source systems, we find that the decisions of log levels might be related to the locations of the logging statements and the messages to be recorded, and log levels that are further apart in order (e.g., *trace* and *error*) tend to have more differences in their characteristics of locations and messages. We then extract syntactic context features (to represent the location information) of the logging statements as well as their log messages, and propose a deep-learning based approach to automatically suggest log levels. Unlike other multi-class classification tasks which consider the classes as independent, log levels have an ordinal nature, i.e., the levels preserve an order among each other. Therefore, we ordinally encode the log levels to capture their ordinal nature.

We evaluate our approach on nine large-scale open source systems and compare the results with two baseline approaches: a state-of-the-art ordinal regression approach from a prior study [16]; and a deep-learning based approach with standard one-hot encoding. We find that, our models trained using the syntactic context feature achieve an average AUC of 80.8, outperforming our models trained using the log message feature (i.e., with an average AUC of 71.5) in suggesting log levels. Combining both features in our approach would lead to the best performance (i.e., with an average AUC of 83.7).

Trained from either the syntactic context feature (i.e., without log message feature) or the combined feature (i.e., with log message feature), our approach outperforms both baseline approaches in all the studied systems. By further studying the results of our approach, we find that the syntactic context and combined features have a similar capability of distinguishing different log levels; while the log message feature may only be useful for specific levels such as *error* and *warn*.

Finally, we evaluate the benefit and applicability of using data from other systems to enlarge the training data. We find that by carefully choosing the training dataset from other systems, the results of our approach can be further improved. In addition, our approach can achieve encouraging results on cross-system suggestions (e.g., on average 93.8% of the accuracy of within-system suggestions), which still outperform the baseline approaches on within-system suggestions.

The contributions of this paper are as follows:

- We propose an automated deep-learning based approach that leverages the ordinal nature of log levels to make suggestions on choosing log levels[1]. Our approach outperforms the existing state-of-the-art approaches in suggesting log levels.
- Our approach have encouraging cross-system suggestion results, which can benefit the systems without long development histories.
- Our manual study results can be leveraged as guidelines in future research on suggesting and improving log levels.

In short, our findings highlight the potentials of leveraging the characteristics of logging statements in suggesting log levels that can help developers make informed logging decisions. Our results also reveal the challenges and future research directions in assisting developers with logging.

**Paper Organization.** Section II discusses the setup and results of manually studying the characteristics of log levels. Section III describes our deep learning approach on suggesting log levels. Section IV presents the evaluation results of our approach by answering three research questions. Section V discusses the threats to the validity of our study. Section VI summarizes the related work. Section VII concludes the paper.

## II. Preliminary study on log levels

### A. An Overview of the Studied Systems

**Studied Systems.** We conduct the study on nine large-scale open source Java systems. Table I shows an overview of the systems. The studied systems are in various sizes (LOC from 97K to 1.5M, and NOL from 0.4K to 5.5K), have high quality logging code, are commonly used in prior log-related studies [17]–[20], and cover various domains (e.g., database systems and search engines).

**Log Level Distribution.** Table I shows the distribution of the log levels in the studied systems. We find that many logging statements are used to show potential issues during system execution (i.e., on average 24.5% are at the *warn* level and 24.4% are at the *error* level). Note that, in modern logging frameworks such as SLF4J, *fatal* level is removed due to its redundancy with other log levels such as *error* [21]. As we found in the studied systems, only Karaf contains some logging statements with *fatal* level and the number is very small (only 0.5%). Therefore, we focus our study on the other log levels. We find that there is also a large proportion of the logging statements that are used for debugging (i.e., 25.5% for *debug*). As mentioned by the instruction of SLF4J, the *trace* level is not recommended since it has a high overlap with the *debug* level [21]. Hence, it may be the reason that some systems have noticeably fewer logging statements at the *trace* level. In general, there are fewer logging statements that show the general system execution (i.e., 17.5% are at the *info* level). Our preliminary findings show that the studied systems have a different distribution of log levels, and the levels are not evenly distributed. Therefore, suggesting log levels accurately either within the same or cross systems is a challenging task.

TABLE I
An overview of the studied systems and their log level distributions (%)

| System | Version | LOC | NOL | Trace | Debug | Info | Warn | Error | Fatal |
|---|---|---|---|---|---|---|---|---|---|
| **Cassandra** | 3.11.4 | 432K | 1.3K | 16.7% | 10.9% | 15.8% | 16.8% | 39.8% | 0.0% |
| **ElasticSearch** | 7.4.0 | 1.50M | 2.5K | 28.5% | 32.4% | 10.0% | 19.2% | 9.9% | 0.0% |
| **Flink** | 1.8.2 | 177K | 2.5K | 1.0% | 30.8% | 26.6% | 23.7% | 17.9% | 0.0% |
| **HBase** | 2.2.1 | 1.26M | 5.5K | 7.4% | 17.3% | 17.1% | 24.4% | 33.8% | 0.0% |
| **JMeter** | 5.3.0 | 143K | 1.9K | 0.7% | 29.9% | 16.9% | 26.5% | 26.0% | 0.0% |
| **Kafka** | 2.3.0 | 267K | 1.5K | 12.9% | 28.5% | 20.4% | 15.3% | 22.9% | 0.0% |
| **Karaf** | 4.2.9 | 133K | 0.8K | 0.9% | 21.9% | 23.1% | 30.0% | 23.6% | 0.5% |
| **Wicket** | 8.6.1 | 216K | 0.4K | 2.2% | 39.3% | 7.6% | 28.5% | 22.4% | 0.0% |
| **Zookeeper** | 3.5.6 | 97K | 1.2K | 2.2% | 18.3% | 19.3% | 35.3% | 24.9% | 0.0% |
| *Average* | —— | 469K | 2.0K | 8.0% | 25.5% | 17.5% | 24.5% | 24.4% | 0.1% |

Note: LOC refers to the lines of code, NOL refers to the number of logging statements.

### B. Investigating Log-level-related Issues

We collect the most recently *resolved* issue reports (from Jan. 2020 to Jul. 2020) in the bug tracking systems of our studied systems, and identify the log-related issue reports by examining if there are changes or patches on logging statements (106 issue reports in total). We then manually examine the changes and the discussions in those issue reports. We find that a large portion (45/106, 42.5%) of the issue reports have changes or discussions on log levels. Specifically, for 23/45 (51.1%) of the log-level-related issue reports, developers suggested changes of log levels on existing logging statements. For 22/45 (48.9%) of the log-level-related issue reports, developers suggested adding new logging statements and mentioned the reasons of the log levels of those newly added logging statements based on their execution point and the messages. In short, the proper choice of log levels is important and is actively considered by developers in both processes of improving existing logging statements and composing new ones. Both the locations and the messages of the logging statements might be important for deciding log levels.

### C. Manually Studying the Characteristics of Log Levels

Prior work [13], [14], [16] found that developers spend significant efforts modifying the levels of existing logging

[1]We share the data of this paper in the repository: https://github.com/SPEAR-SE/ICSE2021_Log_Level_Data.

statements that were inserted previously, and they tend to evaluate the impact of their logging statements and adjust their log levels over time [13]. Motivated by the prior studies and our investigation on log-level-related issue reports, we conduct a manual study to investigate the characteristics of different log levels, in order to better provide supports for developers on deciding log levels. In particular, we study the message and location of a logging statement to investigate if a log level is implicitly or explicitly related to the context information or the log message of the logging statement.

**Manual Study Process.** To prepare the data for our manual study, we first extract the logging statements from the source code using static analysis. We identify the method invocation statements that invoke common logging libraries (e.g., Log4j [12] and SLF4J [21]). Then, for each identified logging statement, we extract its log message (including static message and dynamic variables), verbosity level, and the method that contains the logging statement. In total, we extract 17.6K logging statements from the nine studied systems. Then, we randomly sample 376 out of 17.6K logging statements based on a 95% confidence level and a 5% confidence interval [22]. We apply stratified sampling to ensure the distribution of logging statements from different systems and their log levels in the sampled data is the same as the complete data [23]. Our manual study contains the following three phases:

*Phase I* : We leverage the categories of logging locations and messages that were derived in prior studies [24], [25]. Two authors of this paper (i.e., A1 and A2) use the categories to categorize 100 randomly sampled logging statements collaboratively. During this phase, the categories of logging locations and log messages are revised and refined. In the end, we reused and revised three categories of logging locations and three categories of log messages. We also derived two new categories of logging locations in this phase.

*Phase II* : A1 and A2 independently categorized the rest of the sampled logging statements (276 logging statements) by using the categories derived in Phase I.

*Phase III* : A1 and A2 compared the results from Phase II. Any disagreement of the categorization was discussed until reaching a consensus. No new categories were introduced during the discussion. The results in this phase have a substantial-level of agreement [26] for both of the categorizations of logging location and log message (Cohen's Kappa of 0.82 and 0.88 for logging location and log message, respectively).

**Manual Study Results.** Table II shows the distribution of the categories of logging locations and log messages for different log levels. Each row represents the number of logging statements that belong to each category, and each column represents the number of logging statements with each log level. The percentage in each cell shows the ratio between the logging statements out of the total sampled logging statements with the corresponding log level. Below, we discuss the results by each category.

TABLE II
THE DISTRIBUTION OF THE CATEGORIES OF LOGGING LOCATIONS AND LOG MESSAGES FOR EACH LOG LEVEL

| Category | | Trace | Debug | Info | Warn | Error |
|---|---|---|---|---|---|---|
| Location | **CT** | 2/39 (5.2%) | 4/88 (4.6%) | 12/70 (17.2%) | 37/89 (41.6%) | 52/90 (57.8%) |
| | **LB** | 14/39 (35.9%) | 33/88 (37.5%) | 29/70 (41.4%) | 50/89 (56.2%) | 31/90 (34.5%) |
| | **LP** | 3/39 (7.7%) | 4/88 (4.6%) | 1/70 (1.4%) | 0/89 (0.0%) | 0/90 (0.0%) |
| | **MT** | 10/39 (25.6%) | 14/88 (15.8%) | 6/70 (8.6%) | 1/89 (1.1%) | 3/90 (3.3%) |
| | **OP** | 10/39 (25.6%) | 33/88 (37.5%) | 22/70 (31.4%) | 1/89 (1.1%) | 4/90 (4.4%) |
| Message | **OD** | 24/39 (61.5%) | 42/88 (47.7%) | 49/70 (70.0%) | 27/89 (30.3%) | 1/90 (1.1%) |
| | **VD** | 15/39 (38.5%) | 37/88 (42.1%) | 6/70 (8.6%) | 7/89 (7.9%) | 1/90 (1.1%) |
| | **ND** | 0/39 (0.0%) | 9/88 (10.2%) | 15/70 (21.4%) | 55/89 (61.8%) | 88/90 (97.8%) |

*Categories of Logging Locations v.s. Log Levels*

*Location 1: Catch Clause (CT).* Catch clause is used for capturing the exceptions raised during the execution. As shown in the code snippet below, developers often log the exception information (e.g., the context information of the execution point) in catch clauses [24]. In our manual study, we find that a large portion of the sampled *warn* (37/89, 41.6%) and *error* (52/90, 57.8%) logging statements are in this category. However, there are still a non-negligible number of logging statements that have different log levels. The percentage for the other three levels ranges from 4.6% (4/88 at the *debug* level) to 17.2% (12/70 at the *info* level).

```
/* Location Category 1: Catch Block (CT) */
  } catch (Exception ex) {
    LOG.error("Failed to stop infoServer", ex);
  }
```

*Location 2: Logic Branch (LB).* Logic branch is the code statement that leads to different system execution paths (e.g., *if-else and switch*) [24]. Developers may insert logging statements in the logic branches to help identify the execution path, or record the information in some critical branches. As shown in the code snippet below, developers added a *warn* logging statement to record an unexpected branch execution. We find that the distribution of the five log levels for the logging statements in LB are similar. Each log level has many logging statements in this category, the percentage ranges from 34.5% (31/90 at the *error* level) to 56.2% (50/89 at the *warn* level).

```
/* Location Category 2: Logic Branch (LB) */
  if (logFileReader == null) {
    LOG.warn("Nothing to split in WAL={}", logPath);
    return true;
  } else {
```

*Location 3: Looping Block (LP).* Logging statements in looping blocks (e.g., *for* and *while*) may record the execution state during iterating (e.g., recording the *i*th execution inside a *for* block) or recording variable values as shown in the code snippet below. We do not find any logging statements at the *warn* or *error* level that belong to this category. The logging statements that belong to this category generally have three log levels: 7.7% (3/39) at the *trace* level, 4.6% (4/88) at the *debug* level, and 1.4% (1/70) at the *info* level.

```
/* Location Category 3: Looping Block (LP) */
  while (active) {
    logger.trace("checking jobs [{}]",
          clock.instant().atZone(ZoneOffset.UTC));
    checkJobs();
```

*Location 4: Method Start or End (MT).* Logging statements might reside at the beginning or the end of a method, mostly for recording the program execution state or debugging purposes. For example, the code snippet below logs the event execution time whenever the method is executed. We find that 25.6% (10/39) of the logging statements are at the *trace* level, 15.8% (14/88) are at the *debug* level, and 8.6% (6/70) are at the *info* level. However, logging statements with *warn* and *error* level only have a small portion: 1/89 (1.1%) and 3/90 (3.3%), respectively.

```
/* Location Category 4: Method Start or End (MT) */
public void onEventTime(long timerTimestamp) {
  logger.trace("onEventTime @ {}", timerTimestamp);
```

*Location 5: Observation Point (OP).* We categorize the rest logging locations that do not belong to any of the above-mentioned categories as Observation Point [24]. Logging statements in this category may have various characteristics of logging locations, such as locating before the entry point or after the exit point of a code block to record the execution status (as shown in the code snippet below). We find that a large portion of logging statements that belong to this category (from 25.6% to 37.5%) is at the *trace*, *debug*, and *info* level; while only 1.1% (1/89) and 4.4% 4/90 are at the *warn* and *error* level, respectively.

```
/* Location Category 5: Observation Point (OP) */
  final BinaryInMemorySortBuffer buffer =
       currWriteBuffer.buffer;
  LOG.debug("Retrieved empty read buffer " +
       currWriteBuffer.id + ".");
  long occupancy = buffer.getOccupancy();
  if (!buffer.write(current)) {
```

### *Categories of Log Messages v.s. Log Levels*

*Message 1: Operation Description (OD).* Log messages in this category summarize the actions or intentions of its surrounding code [25]. Logging statements with this kind of log message could be placed before, inside, or after the execution point to record the status of an upcoming, ongoing, or a completed operation. As shown in the example below, an *info* logging statement logs the closing of a connection. We find that most of the *info* logging statements (49/70, 70.0%) are in this category. There are also a large portion of *trace* (24/39, 61.5%) and *debug* (42/88, 47.7%) logging statements in this category. For *warn* and *error* level, 30.3% (27/89) and 1.1% (1/90) of the logging statements belong to this category.

```
/* Message Category 1: Operation Descripion (OD) */
  connectionTracker.closeAll();
  logger.info("Stop listening for CQL clients");
```

*Message 2: Variable Description (VD).* Variable description records the value of a variable during execution [25]. As shown in the example below, a *trace* logging statement is placed after defining the variable *parameterMap* to record its value. We find that many logging statements at the *trace* (15/39, 38.5%) and *debug* (37/88, 42.1%) level belong to this category. For other levels, the percentage is noticeably smaller (from 1.1% at the *error* level to 8.6% at the *info* level).

```
/* Message Category 2: Variable Description (VD) */
  Map<String, List<String>> parameterMap =
       request.getParameterMap();
  LOG.trace("parameterMap: {}", parameterMap);
  if (parameterMap != null) {
```

*Message 3: Negative Execution Behavior Description (ND).* During the system runtime, some unexpected execution behaviors may happen (e.g., an exception, or a failure). Logging statements are often inserted into these unexpected execution points to record the related information. Hence, developers can then be aware of the problem and fix the issue. We consider log messages as this category if they describe an unsuccessful attempt or an unexpected situation, with some specific negative words (e.g., fail, exception, unable). We find that most of the *error* (88/90, 97.8%) and a large number of *warn* (55/89, 61.8%) logging statements are in this category. For *info* and *debug* level, there are 21.4% (15/70) and 10.2% (9/88) of the logging statements in this category, respectively. We do not find logging statements at the *trace* level that belong to this category.

```
/* Message Category 3: Negative Execution Behavior
     Description (ND) */
  if (tokensIndex.isAvailable() == false) {
    logger.warn("failed to get access token [{}]",
         tokenId);
    listener.onResponse(null);
  } else {
```

### *Summary of the Manual Study Findings*

As we found in our manual study, the information of logging locations and log messages may be related to the decision of log levels. For example, we find that developers are more likely to set the log level to *warn* or *error* if the logging statement resides in catch blocks (category CT). Moreover, if the logging statements reside at the beginning or end of a method (category MT), the log levels are more likely to be *trace* or *debug*. Similarly, logging statements with certain types of log messages, such as the category VD (variable description), are more often set to *trace* or *debug* level. Log levels that are further apart in order (e.g., *trace* and *error*) tend to have more different characteristics to distinguish. Our findings shed light on the relationship between log levels and the categories of logging location and log messages, as well as the ordinal nature of log levels, that may be further leveraged to assist developers in determining log levels.

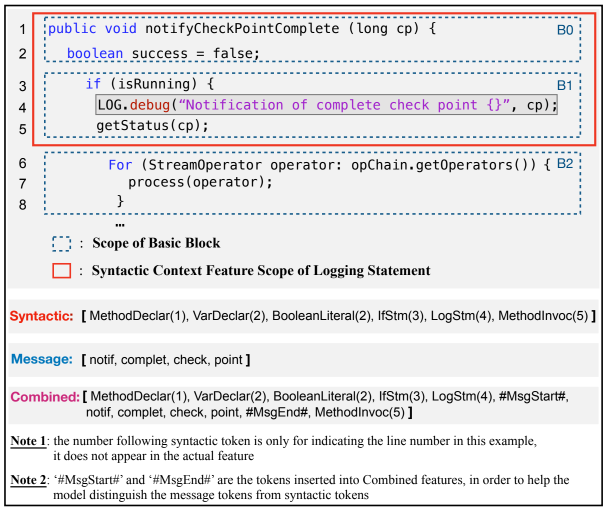
We find that log levels that are further apart in order tend to have more different characteristics of logging locations and log messages. Locations and messages of logging statements, as well as the ordinal nature of log levels might be leveraged to help decide log levels.

## III. Automatically Suggesting Log Levels

Inspired by our manual study findings, in this section, we propose an approach that automatically suggests log levels. We formulate the process of suggesting log levels as a multi-class classification problem. Given the information of an existing or a potential new logging statement (i.e., the structural information, or the log message, or both), we apply deep learning models to suggest which level to use. Below, we discuss how we extract the features and the framework of our deep learning approach for suggesting log levels.

### *A. Feature Extraction*

For each logging statement, we extract three types of features: syntactic context features (simplified as Syn in the rest of paper), log message features (Msg), and combined features of syntactic context and log messages (Comb).



```
1  public void notifyCheckPointComplete (long cp) {                  B0
2    boolean success = false;
3      if (isRunning) {                                              B1
4        LOG.debug("Notification of complete check point {}", cp);
5        getStatus(cp);
6          For (StreamOperator operator: opChain.getOperators()) { B2
7            process(operator);
8          }
           ...
```

Syntactic: [ MethodDeclar(1), VarDeclar(2), BooleanLiteral(2), IfStm(3), LogStm(4), MethodInvoc(5) ]

Message: [ notif, complet, check, point ]

Combined: [ MethodDeclar(1), VarDeclar(2), BooleanLiteral(2), IfStm(3), LogStm(4), #MsgStart#, notif, complet, check, point, #MsgEnd#, MethodInvoc(5) ]

Note 1: the number following syntactic token is only for indicating the line number in this example, it does not appear in the actual feature

Note 2: '#MsgStart#' and '#MsgEnd#' are the tokens inserted into Combined features, in order to help the model distinguish the message tokens from syntactic tokens

Fig. 1. An example of the syntactic, log message, and combined features we extracted for each logging statement

**Syntactic Context Features.** We extract the syntactic context feature that represents the location information of a logging statement. Specifically, we parse the Abstract Syntax Tree (AST) of the source code and extract the AST nodes that are related to the control flow of the code to capture the structural information (e.g., IfStatement and CatchClause). We exclude the AST nodes that do not contain structural information of the code, such as SimpleName (i.e., identifier name) and SimpleType (i.e., identifier type). We also exclude AST nodes that are related to log guards (e.g., `if(isTraceEnabled)`). For each logging statement, we count the occurrence of each AST node from the start of the method, to the end of the basic block in which the logging statement resides. We analyze the AST nodes from the beginning of a method since the nodes represent the syntactic context of the logging statement (e.g., the logical flow of the method). As we found in Section II, such syntactic context have a certain relationships with log levels. We choose to extract the features based on basic blocks since they represent a sequence of code statements where there is no branching in between (i.e., no other structural information that can affect the decision of the level of a logging statement in the block). Finally, we obtain a set of tokens (i.e., AST nodes) for each logging statement that represents the syntactic feature of the logging statement. Figure 1 shows an example of the syntactic context feature that we extract for the logging statement on line 4.

**Log Message Features.** We extract the log message features from the textual information inside the logging statements. We exclude the dynamic variables, since many variable names in logging statements are not composed of natural language words [27] (e.g., variable *cp* in the logging statement in Figure 1, which is abbreviated from "check point"). For the static message in each logging statement, we first split the words using space and camel cases. We then follow common text pre-processing techniques [28]: remove the punctuation, convert the words into lower case, filter the common English words [29] and apply stemming [30] on the filtered words [20], [27], [31]. At the end of this process, we obtain a set of log message tokens, which represents its log message feature, for each logging statement.

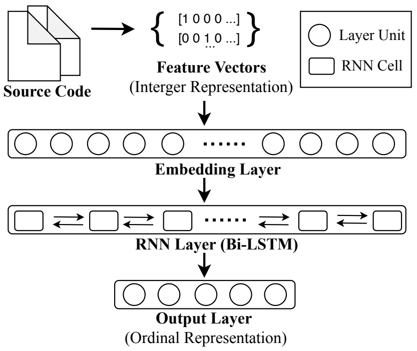


Fig. 2. Overall framework of our approach

**Combined Features.** As we found in Section II, both the logging locations and log messages may have a certain relationship with the log levels, as they capture different aspects of a logging statement. Therefore, we combine both the syntactic information and the log message, by following an approach that is similar to prior studies [32], [33]. For each logging statement, we add the log message feature to the syntactic feature and preserve their actual order in the source code (i.e., the log message feature is added to the place that the logging statement appears in the source code). We then add a special token at the beginning and the end of the log message feature to help the model distinguish it with syntactic information. Finally, we obtain a set of tokens for each logging statement that represents the combined feature of the logging statement. Figure 1 shows an example of how do we combine the syntactic context feature and log message feature for the logging statement in line 4.

**Ordinally Encoding Log Levels.** One-hot encoding is widely used by prior studies for multi-class classification problems [27], [34], [35]. However, log levels, by nature, have an ordinal relationship. For example, if the system is configured to run and record *debug* logs, the system would also enable logging statements that are at the *info*, *warn*, and *error* levels and record logs in those levels. Therefore, we ordinally encode the log levels to preserve such ordinal relationship when suggesting log levels. Table III shows the comparison between the vectors of each log level that are ordinally encoded and encoded by standard one-hot encoding. Our encoding preserves the ordinal characteristics of log levels, where when a system is configured to record a certain log level (e.g., *info*), the system would also record all logs that have a higher log level (e.g., *warn* and *error*).

### *B. Deep Learning Framework and Implementation*

**Overall Architecture.** Figure 2 shows the overall architecture of our approach. The deep learning framework contains an embedding layer, an RNN layer, and an output layer. Given the syntactic features, log message features, or combined features of logging statements, the embedding layer learns the relationship among the input vectors and transform each vector to a distributed representation based on probability. We then use a recurrent neural network (RNN) layer to learn the relationship between the log level and the embedded vectors returned from the embedding layer. After that, the output layer gives an ordinal vector as the suggestion result. Finally, we map the ordinal vector returned from the output layer to a real log level as the final result. Below, we discuss the details of each component of our approach.

**Embedding Layer.** Our extracted features (i.e., Syn, Msg, and Comb) are represented in the form of vectors. Each dimension represents the unique tokens of the corresponding feature (e.g., the types of AST node in Syn), and each element represents the number of occurrences of the token for each logging statement. We then feed the feature vectors into the embedding layer. The embedding layer captures the linear relationships among the tokens in the feature vectors, and outputs the probabilistic representations of the vectors (i.e., word embeddings [36]). In other words, word embeddings learn the similarities among the tokens to create a more concise representation of the features [37]–[39].

**RNN Layer.** We model the source code and log message as sequential data (i.e., the order of the tokens that appear in the source code is preserved) by following prior studies [40]–[43]. We employ a layer of Bidirectional Long Short Term Memory (Bi-LSTM) in the deep learning model, which is widely used by prior studies to process source code and natural language [31], [44]. Bi-LSTM is a variant of RNN that concatenates the outputs of two RNNs, one processing the sequence of input vector from the beginning to the end, the other one from the end to the beginning. Each RNN is composed of recurrent units including a memory cell and gate mechanisms to preserve long term dependencies of the given input. While training the model, we encode the log level of a logging statement into its ordinal representation (as discussed in Subsection III-A).

**Output Layer.** We then use a five-dimension dense layer as the output layer. Specifically, the output layer takes the high-dimensional output vectors from the previous layer (i.e., the RNN layer) to the five neurons in this layer. Each of the five neuron represents one number in our ordinally encoded vector of log level. Then each neuron gives the result of the corresponding number (i.e., the probability of this digit to be 1) in the vector. After that, we accept the output vector and map the vector into an actual log level as the final suggestion result. For example, if the returned vector from the output layer is [1.0, 0.8, 0.6, 0.3, 0.1], we check each probability value from the start to the end of the vector. If a probability is larger than 0.5, the number is mapped to 1. If a probability that is smaller than 0.5 is encountered, the rest numbers will be mapped to 0. In the above-mentioned example, the output vector will be mapped to [1, 1, 1, 0, 0], as discussed in Subsection III-A, which is an *info* level.

**Implementation and Training** We use Keras [45] to implement our deep learning framework. We use Skip-gram from Word2vec [46] in the embedding layer and set the dimension to 100 by following prior work [27]. We obtain the word embeddings for each type of features (i.e., syntactic context, log message, and combined features) separately. For the RNN layer, we set the number of units (i.e., the dimension of hidden states) as 128 and attach a dropout layer with a 0.2 dropout rate, in order to reduce the potential impact of overfitting on the trained system [47]–[49]. For each training process, we set the number of epochs as 100 and the batch size as 24 [27]. Since the model learns and predicts on each digit of the ordinally encoded vector, we use sigmoid as the activation function and use binary cross entropy as the loss function. Note that the distribution of log levels is noticeably different (e.g., on average, only 8.0% of the logging statements are in *trace* while 24.4% of the logging statement are in *error* level), as discussed in Section II. Hence, we apply stratified random sampling [23] while splitting the training, validation, and testing data to ensure the sampled data set has the same distribution of log levels as the original data.

TABLE III
A COMPARISON BETWEEN THE VECTORS OF LOG LEVELS THAT ARE ORDINALLY-ENCODED AND ONE-HOT ENCODED

| | **Ordinally Encoded** | **One-hot Encoded** |
|---|---|---|
| **Trace** | [1, 0, 0, 0, 0] | [1, 0, 0, 0, 0] |
| **Debug** | [1, 1, 0, 0, 0] | [0, 1, 0, 0, 0] |
| **Info** | [1, 1, 1, 0, 0] | [0, 0, 1, 0, 0] |
| **Warn** | [1, 1, 1, 1, 0] | [0, 0, 0, 1, 0] |
| **Error** | [1, 1, 1, 1, 1] | [0, 0, 0, 0, 1] |

## IV. EVALUATION

### A. Evaluation Metrics

We use Accuracy and Area Under the Curve (AUC), which are widely used by prior multi-class classification studies, to evaluate our approach [16], [27]. According to the ordinal nature of log levels, we also propose a new metric, Average Ordinal Distance Score (AOD), to measure the average distance between the actual level and the suggested level.

**Accuracy.** Similar to the usage in prior classification studies [27], [42], Accuracy in our study is the percentage of correctly suggested log levels out of all the suggestion results. A higher accuracy means a model can correctly suggest the log levels for more logging statements. As a reference, the accuracy of a 5-category random guess is around 20%.

**Area Under the Curve (AUC).** AUC is the area under the ROC (receiver operating characteristic) curve that plots the true positive rate against the false positive rate, which evaluates the ability of a model in discriminating different classes. AUC ranges between 0 and 1: a high value for the AUC indicates a high discriminative ability of a model; an AUC lower than 0.5 indicates a performance that is not better than random guessing. Following prior work [16], we use a multiple-class version of the AUC defined by Hand et al. [50]. The AUC gives us the insight about how well the model can discriminate different log levels, e.g., how likely a model is able to predict an actual *info* level as *info* (i.e., true positive), rather than predict an actual *debug* level as *info* (i.e., false positive).

**Average Ordinal Distance Score (AOD).** The prior two metrics consider different log levels as independent classes (i.e.,

TABLE IV
THE RESULTS OF SUGGESTING LOGGING LEVELS USING SYNTACTIC CONTEXT (Syn), LOG MESSAGE (Msg), AND A COMBINATION OF BOTH (Comb), COMPARED WITH ORDINAL REGRESSION (OR) AND ONE-HOT ENCODING NEURUAL NETWORK (OEN)

| | **Accuracy** | | | | | **AUC** | | | | | **AOD** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Systems | Syn | Msg | Comb | OR | OEN | Syn | Msg | Comb | OR | OEN | Syn | Msg | Comb | OR | OEN |
| **Cassandra** | **53.7** | **52.6** | ***60.6*** | 43.2 | 39.9 | **78.8** | **77.0** | ***84.2*** | 75.6 | 70.9 | **78.9** | **77.9** | ***80.5*** | 70.3 | 65.6 |
| **Elasticsearch** | **51.9** | 40.4 | ***57.7*** | 49.8 | 37.8 | **77.9** | 66.4 | ***81.3*** | 75.8 | 72.6 | **77.6** | 69.1 | ***80.2*** | 76.8 | 41.7 |
| **Flink** | **52.5** | 35.5 | ***65.2*** | 50.1 | 42.0 | **78.2** | 69.9 | ***85.1*** | 74.1 | 73.5 | **78.7** | 72.4 | ***83.8*** | 78.5 | 43.9 |
| **HBase** | **55.9** | 50.7 | ***60.3*** | 51.0 | 49.5 | **83.1** | 74.3 | ***84.2*** | 79.4 | 78.3 | **81.4** | 72.8 | ***81.7*** | 78.9 | 62.7 |
| **JMeter** | **55.1** | 52.1 | ***62.3*** | 53.9 | 47.2 | **83.5** | 79.3 | ***83.9*** | 80.7 | 76.5 | **82.3** | 77.2 | ***80.9*** | 78.9 | 56.2 |
| **Kafka** | **50.7** | 38.5 | ***51.8*** | 42.3 | 41.8 | **78.7** | 71.5 | ***79.5*** | 74.1 | 69.2 | **76.9** | 68.6 | ***77.5*** | 72.2 | 59.4 |
| **Karaf** | **56.5** | 30.3 | ***67.2*** | 49.0 | 30.3 | **84.3** | 67.2 | ***85.6*** | 83.2 | 68.1 | **80.6** | 67.2 | ***81.6*** | 76.8 | 30.9 |
| **Wicket** | **57.3** | 28.1 | ***63.8*** | 46.1 | 39.0 | **83.1** | 61.6 | ***85.0*** | 80.7 | 76.4 | **78.9** | 62.1 | ***79.3*** | 67.8 | 54.5 |
| **Zookeeper** | **52.8** | 50.1 | ***60.9*** | 41.3 | 35.1 | **79.6** | 76.1 | ***84.8*** | 79.1 | 69.2 | **78.8** | 73.0 | ***82.0*** | 77.0 | 36.8 |
| *Average* | **54.0** | 42.0 | ***61.1*** | 47.4 | 40.3 | **80.8** | 71.5 | ***83.7*** | 78.8 | 72.7 | **79.3** | 71.1 | ***80.8*** | 75.2 | 50.2 |

Note: The number that is higher than both of the baselines is marked in **bold**, the best result is marked in ***italic-bold***.

the ordinal nature of log levels, as discussed in Section III, is not considered). Hence, we propose Average Ordinal Distance Score (AOD) which measures the average distance between the *actual* log level and the *suggested* log level for each logging statement. It is computed as:

$$AOD = \frac{\sum_{i=1}^{N} (1 - Dis(a_i, s_i)/MaxDis(a_i))}{N},$$

where $N$ is the total number of logging statements in the results. For each logging statement and its suggested log level, *Dis(a, s)* is the distance between the *actual* log level $a_i$ and the *suggested* log level $s_i$ (e.g., the distance between *error* and *info* is 2). *MaxDis(a)* is the maximum possible distance of the *actual* log level $a_i$. For example, the maximum possible distance for *trace* is 4 (i.e., from *trace* to *error*), for *info* is 2 (i.e., from *info* to *trace* or *error*). A higher AOD indicates *suggested* log levels are closer to their *actual* log levels.

## B. Case Study Results

**RQ1: How effective is our approach in suggesting log levels?**

**Motivation.** As we found in the manual study, the decision of log level may be related to the syntactic information in the code and the log message. In this RQ, we want to evaluate the performance of our deep learning models trained using each of the three features (i.e., syntactic context, log message, and combined, as described in Section III-A).

**Approach.** We first apply stratified random sampling [23] to split the input data into training set (60%), validation set (20%), testing set (20%) [27], [42], and ensure each of the sampled datasets has the same distribution of log levels as the original data. We compare our approach with two baselines described below. We then train our deep learning framework and the two baselines on the training data using each of the three features. Below, we describe the two baselines in details.

*Baseline 1: Ordinal Regression (OR) model.* We use ordinal regression (OR) models [51] to suggest log levels by following a prior study [16]. OR considers the orders of the log levels (e.g., *error* is more severe than *info*) when training the model and predicting the log level given a new logging statement. In this work, we migrate the OR approach to our problem context: suggesting the log level of each logging statement in the static code. We consider all the metrics used in the prior work [16] except those related to code changes, as the code change related metrics are irrelevant in our context: we suggest the log level of each logging statement in the static code. Besides, prior work finds that the influence of the metrics related to the code changes is negligible [16].

*Baseline 2: One-hot Encoding RNN (OEN).* As discussed in Section III, the standard one-hot encoding treats all the classes as independent classes without considering the ordinal relation among them. In order to understand the effectiveness of our encoding on log levels, we would like to compare the performance of the models using our ordinally encoded log level vectors with the models using standard one-hot encoded log level vectors. Different from our approach that uses sigmoid as the activation function and binary cross entropy as the loss function to predict the value of each number in the ordinally encoded vector (as discussed in Section III), to adopt standard one-hot encoding, we change the activation function to softmax and the loss function to categorical cross entropy [34], [35]. Hence, the goal of the baseline model is to predict the one-hot encoded vector (as shown in Table III) which can be mapped back to a log level. Similar to our approach, we train the baseline on the Syn, Sem, and Comb features, respectively.

**Results and Discussions.**

**Our approach can effectively suggest log levels for the studied systems. Our best models (i.e., using the combined feature) achieve an average AUC of 83.7.** Table IV presents the results of our models trained using the syntactic feature (Syn), the log message feature (Msg), and the combined feature (Comb). Table IV shows that the models trained using the Syn feature perform better than the models using the Msg feature in terms of all the three evaluation metrics. Specifically, the average accuracy, AUC, and AOD of the models trained using the Syn feature are 54.0, 80.8, and 79.3, respectively; while the average accuracy, AUC, and AOD of the models trained using the Msg feature are only 42.0, 71.5, and 71.1. More importantly, for all the three evaluation metrics, the models trained using the Comb feature have better results than the models trained only using Syn or Msg. Specifically, on average, the accuracy, AUC, and AOD of the models trained using Comb are 61.1, 83.7, and 80.8, respectively. Our results show that the Syn and Msg features both provide valuable information that can complement each other in our models.

**Our approach outperforms the two baseline approaches**

TABLE V
THE DISTRIBUTION OF INCORRECTLY SUGGESTED LOG LEVELS FOR EACH ACTUAL LOG LEVEL (THE FIRST COLUMN)

| | **Syntactic Context** | | | | | **Log Message** | | | | | **Combined** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Trace | Debug | Info | Warn | Error | Trace | Debug | Info | Warn | Error | Trace | Debug | Info | Warn | Error |
| **Trace** | — | **53.8** | 25.4 | 16.7 | 4.1 | — | **47.6** | 30.1 | 20.7 | 1.6 | — | **52.4** | 21.1 | 18.7 | 7.8 |
| **Debug** | 5.4 | — | **53.7** | 33.6 | 7.3 | 7.3 | — | **48.1** | 41.7 | 2.9 | 7.5 | — | **43.6** | 32.6 | 16.3 |
| **Info** | 3.6 | 38.8 | — | **48.9** | 8.7 | 6.6 | 37.9 | — | **52.2** | 3.3 | 8.6 | **43.0** | — | 38.1 | 10.3 |
| **Warn** | 1.6 | 27.8 | 28.7 | — | **41.9** | 3.2 | 36.4 | 22.4 | — | **38.0** | 4.1 | 20.5 | 27.0 | — | **48.4** |
| **Error** | 0.9 | 7.7 | 20.7 | **70.7** | — | 0.4 | 8.2 | 12.9 | **78.5** | — | 3.9 | 12.3 | 10.4 | **73.4** | — |

Note: For each feature and each actual log level, the highest percentage of incorrectly suggested log level is marked in **bold**.

**(OR and OEN).** For the baselines, due to the limitation of space, we only discuss the results of the models trained using the comb feature, which lead to the best results among the three features. For the the results of two baselines, the average accuracy are 47.4 for OR and 40.3 for OEN, the average AUC are 78.8 for OR and 72.7 for OEN, and the average AOD are 75.2 and 50.2, respectively. For every system, our models using Syn or Comb features always outperform the two baselines in the three evaluation metrics (as shown in Table IV). The results demonstrate the higher capability of our neural networks with ordinally encoded log levels than the ordinal regression and the standard one-hot encoding in suggesting log levels.

Our approach outperforms the two baseline approaches in suggesting log levels. In particular, our approach achieves the best performance when both the syntactic and log message features are considered.

**RQ2: What is the performance of our approach on different log levels?**

**Motivation.** In RQ1, we find that our approach can effectively suggest the log level of a logging statement, and that the models trained using the syntactic, message, and combined information show different performance. However, for the logging statements with different log levels, choosing an inappropriate log level may have different costs. For example, choosing the *error* level for an *info* message may be worse (i.e., cause user confusion [13]) than choosing the *info* level for a *debug* message. Besides, different stakeholders may be more interested in certain log levels. For example, operators may be most interested in the *warn* and *error* levels which need their immediate actions; while developers doing debugging activities may be most interested in the *debug* level. Therefore, in this RQ, we further investigate the performance of our approach in providing suggestion for each log level.

**Approach.** We first analyze the overall performance of our models for each log level. We group the logging statements in our test datasets by their *actual* log level. Then, for each group of *actual* log level, we measure the accuracy of our approach for suggesting the log levels. We then investigate how our models *mis-classify* each log level by computing the distribution of the incorrectly suggested log levels. In this RQ, we train the models and analyze the results of the three features (i.e., Syn, Msg, or Comb), respectively.

**Results and Discussions.**

**The syntactic and combined features show more consistent performance than the log message feature among suggesting different log levels.** For each log level and each feature, we present the results by showing the average accuracy of the models trained using different systems. Figure 3 shows the accuracy of the trained models using syntactic context feature (red bar), log message feature (blue bar), and combined features (purple bar) for each log level. Overall, the syntactic context and combined features have a similar trend on the results for different levels, while log message features have a notable difference. The log message feature has a relatively high accuracy on suggesting the *warn* and *error* levels, but has a very low accuracy on other levels (ranges from 8.9% to 16.8%). The potential reason might be that, *warn* and *error* level might contain some specific words that can be used to distinguish them from other levels. As we found in Section II, 61.8% and 97.8% of the log messages at *warn* and *error* level describe negative execution behaviors. However, syntactic and combined feature also achieve relatively good results on these two levels (range from 58.8% to 59.6% for *warn* level, and from 52.5% to 77.4% for *error* level). Both the syntactic and combined features also have reasonable results on suggesting *debug* and *info* levels (range from 42.6 to 55.3 accuracy).

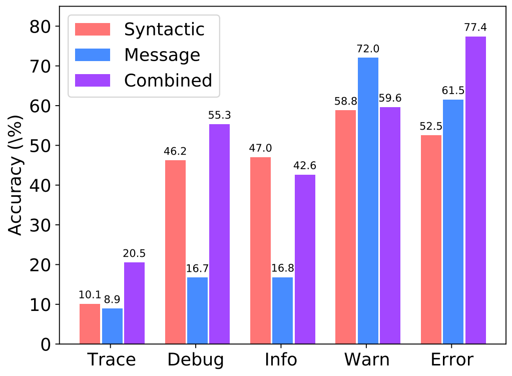


Fig. 3. The accuracy of our approach on each log level

**Most of the incorrectly suggested log levels provided by our approach are close to their *actual* log levels.** Table V presents the distribution of incorrectly *suggested* log levels for each *actual* log level (the first column, marked in bold). All the numbers are the percentage of an incorrectly *suggested* log level over all the incorrect suggestions for each *actual* log level. Overall, there is only a small portion of logging statements that are incorrectly suggested as *trace* level (range from 0.4% to 8.6% across all the three features). In comparison, most of the incorrect suggestions on *error* logging statements are suggested as *warn* level (over 70% for all the three features). Reversely, many *warn* logging statements are incorrectly suggested as *error* level (which is also the most common incorrectly *suggested* log level). We find that for each feature and each *actual* log level, the most common incorrect suggestions are one of their neighbouring log levels (i.e., the closest log levels). In particular, some log levels (e.g., *warn* and *error* levels) might be hard to distinguish. Future studies could conduct in-depth investigations on more characteristics of different log levels and help provide a more accurate suggestion correspondingly.

TABLE VI
THE RESULTS OF COMPARING ENLARGING TRAINING DATA (RQ3-A) ON SYNTACTIC (S-enlarge.) AND COMBINED FEATURE (C-enlarge.) AND CROSS-PROJECT PREDICTION (RQ3-B) ON SYNTACTIC (S-cross.) AND COMBINED FEATURE (C-cross.) WITH THE WITHIN PROJECT PREDICTION IN RQ1

| | **Accuracy** | | | | **AUC** | | | | **AOD** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Systems | S-enlarge | C-enlarge | S-cross | C-cross | S-enlarge | C-enlarge | S-cross | C-cross | S-enlarge | C-enlarge | S-cross | C-cross |
| **Cassandra** | 58.5 (+4.8) | 63.5 (+2.9) | 45.9 (85.5%) | 58.9 (97.2%) | 79.7 (+0.9) | 84.9 (+0.7) | 74.6 (94.7%) | 82.7 (98.2%) | 82.1 (+3.3) | 85.2 (+4.7) | 76.3 (96.7%) | 80.1 (99.5%) |
| **Elasticsearch** | 41.7 (-10.2) | 49.1 (-8.6) | 47.3 (91.1%) | 55.7 (96.5%) | 75.2 (-2.7) | 76.9 (-4.4) | 75.1 (96.4%) | 80.2 (98.6%) | 73.3 (-4.3) | 78.1 (-2.2) | 77.5 (99.8%) | 78.4 (97.8%) |
| **Flink** | 54.4 (+1.9) | 66.1 (+0.9) | 45.2 (86.1%) | 63.7 (97.7%) | 81.5 (+3.3) | 85.5 (+0.4) | 74.4 (95.1%) | 83.5 (98.1%) | 78.8 (+0.1) | 83.9 (+0.1) | 76.8 (97.6%) | 82.3 (98.2%) |
| **HBase** | 57.3 (+1.4) | 64.0 (+3.7) | 40.3 (72.1%) | 55.2 (91.5%) | 84.2 (+1.1) | 85.3 (+0.9) | 72.5 (87.2%) | 80.2 (95.2%) | 82.0 (+0.6) | 84.1 (+2.4) | 73.9 (90.8%) | 76.7 (93.9%) |
| **JMeter** | 56.5 (+1.4) | 63.7 (+1.4) | 44.3 (80.4%) | 53.6 (86.0%) | 84.0 (+0.5) | 84.6 (+0.7) | 73.8 (88.4%) | 76.8 (91.5%) | 82.9 (+0.6) | 83.8 (+2.9) | 75.6 (91.9%) | 76.3 (94.3%) |
| **Kafka** | 51.1 (+0.4) | 52.8 (+1.0) | 45.7 (90.1%) | 50.8 (98.1%) | 79.3 (+0.6) | 80.2 (+0.7) | 74.8 (95.0%) | 76.8 (96.6%) | 77.5 (+0.6) | 78.9 (+1.4) | 75.2 (97.8%) | 75.9 (97.9%) |
| **Karaf** | 57.9 (+1.4) | 68.9 (+1.7) | 45.3 (80.2%) | 62.1 (92.4%) | 85.1 (+0.8) | 86.2 (+0.6) | 74.7 (88.6%) | 83.8 (97.9%) | 82.6 (+2.0) | 83.5 (+1.9) | 77.1 (95.7%) | 81.2 (99.5%) |
| **Wicket** | 58.9 (+1.6) | 65.9 (+2.1) | 45.1 (78.7%) | 58.3 (91.4%) | 83.8 (+0.7) | 86.1 (+1.1) | 74.6 (89.8%) | 81.0 (95.3%) | 80.0 (+1.1) | 84.3 (+5.0) | 76.7 (97.2%) | 78.8 (99.4%) |
| **Zookeeper** | 54.5 (+1.7) | 61.8 (+0.9) | 45.0 (85.2%) | 57.8 (94.9%) | 80.3 (+0.7) | 85.6 (+0.8) | 73.5 (92.3%) | 79.7 (94.0%) | 79.9 (+1.1) | 83.3 (+1.3) | 76.1 (96.6%) | 79.8 (97.3%) |
| *Average* | 54.5 (+0.5) | 61.8 (+0.7) | 44.9 (83.1%) | 57.3 (93.8%) | 81.5 (+0.7) | 83.9 (+0.2) | 74.2 (91.8%) | 80.5 (96.2%) | 79.9 (+0.6) | 82.8 (+2.0) | 76.1 (96.0%) | 78.8 (97.5%) |

Note: The +/- number after each data in the columns of B-enlarge and C-enlarge indicates the improve or decrease compared with within-system prediction in RQ1. The percentage in the columns of B-cross and C-cross represents the ratio against the results of within-system prediction in RQ1.

The syntactic context and combined features show more consistent capability in making suggestion among different log levels, while the log message feature may only provide helpful suggestion on specific levels (e.g., *warn* and *error*). Many of the incorrectly suggested log levels are close to their *actual* log levels. Future work could investigate opportunities that leverage the characteristics of different log levels to distinguish log levels that are close in order.

**RQ3: Can our approach benefit from transfer learning?**

**Motivation.** The success of deep neural networks often requires a large dataset in order to provide sufficient information for training [52], [53]. However, as presented in Section II, the amount of logging statements in the studied systems ranges from 0.4K to 5.5K, i.e., small datasets compared to other areas, such as computer vision, where these deep neural networks are extensively leveraged [54]. Moreover, different from mature systems with a long period of development and maintenance history, new software systems may not have enough existing logging statements to train a deep neural network. Transfer learning techniques are often used to address the challenge of limited dataset [55]. In particular, one may use data from other projects to complement the existing dataset to train a better model. In this RQ, we investigate whether using transfer learning among different studied systems can benefit our approach. In particular, we study two sub-RQs:

**RQ3-A:** Can we improve the performance of our approach by including more training data from other studied systems?

**RQ3-B:** How accurate is our approach in cross-system suggestions?

**Approach.** We choose to study the use of transfer learning with Syn and Comb features of our approach, since both outperform the baselines, as discussed in RQ1. Below, we describe the approach of each sub-RQ.

*RQ3-A:* We enlarge the dataset by combing the data from all the studied systems. For each system, we follow stratified sampling to split the data into training data (60%), validation data (20%), and testing data (20%). We then merge the training data from all studied systems and train a deep learning model, while using the 20% validation data set (combined from every studied system) to validate the model during the training process. Finally, we apply the model trained using the enlarged dataset separately on the testing data of each studied system.

*RQ3-B:* For each target system, we combine the data from the remaining eight systems together and apply stratified sampling to split 80% of the data combined from the eight systems as training data, and 20% as validation data. We then use the complete data of the target system as the testing data and apply the model trained using the combined data from the other eight studied systems.

**Results and Discussions.**

*RQ3-A:* **Our approach can benefit from the enlarged training data from other systems.** Table VI shows the results of enlarging the training set using the syntactic context features (S-enlarge) and the combined features (C-enlarge). The +/- number after each data indicates the improve or decrease compared to within-system suggestion in RQ1. Overall, for both of the two features, the performance is improved in eight of the studied systems on all of the evaluation metrics. Specifically, for syntactic context features (i.e., S-enlarge in Table VI), the improvement of accuracy ranges from 0.4 in Kafka to 4.8 in Cassandra. The average AUC and AOD also improve by 0.7 and 0.6, respectively. For combined features (i.e., C-enlarge in Table VI), the improvement of accuracy ranges from 0.9 in Flink and Zookeeper, to 3.7 in HBase. The average AUC and AOD also improve by 0.2 and 2.0, respectively. On the other hand, the performance in Elasticsearch is decreased after enlarging the training data from other systems (accuracy decreases by 10.2 on S-enlarge, and by 8.6 on C-enlarge). As shown in Table I, Elasticsearch has considerably different log level distribution compared to other systems. In particular, there exist considerably more *trace* level logging statements than other systems (28.5% versus 5.5%); while much fewer *error* level logging statements (9.9% versus 26%). Hence, the data from other systems may not be able to complement the data from Elasticsearch in the model training. Our finding shows that, while enlarging the training data may improve suggestion performance, practitioners should carefully and tactically choose the data when enlarging the training set.

*RQ3-B:* **Our approach achieves encouraging results in cross-system log level suggestions.** Table VI shows the results of cross-system suggestions using the syntactic context features (S-cross) and the combined features (C-cross). The percentage following each number represents the ratio of the corresponding evaluation metric against the results of within-system prediction in RQ1. For example, the accuracy of C-cross in Cassandra is 58.9. Compared with the original within-system accuracy of Comb in Cassandra, i.e., 60.6, the accuracy ratio of C-cross against Comb in Cassandra is 97.2% (58.9/60.6).

Overall, the cross-system suggestions achieve 83.1% accuracy on average for S-cross compared to Syn in RQ1, and achieve 93.8% accuracy on average for C-cross compared to Comb in RQ1. We also find that the results of cross-system suggestions on combined features are still higher than the results of the two baselines in RQ1. In other words, even with cross-system suggestions, our approach can still outperform the two baseline approaches that are trained and tested with data from the same system.

Our approach can benefit from transfer learning. By enlarging the training set, the performance of our approach can be improved in eight out of nine studied systems. Our approach also has an encouraging performance for cross-system log level suggestions, which still outperforms the within-system suggestions by the baseline approaches.

## V. Threats to Validity

**Construct Validity.** To mitigate the fluctuation caused by different testing data set, we follow prior studies to split the training, validation, and testing data [27], [42] and apply stratified random sampling [23], [27], [42] to ensure each randomly sampled data set has the same distribution of log levels as the original data. Our approach presumes that the training data has high-quality source code and follows good logging practice. However, there is no "golden rule" for how to write logging statements, which may affect the stability of logging statements [56], [57]. To mitigate this threat, we choose nine well-maintained, large-scale systems across various domains, with different sizes to conduct our study. They are commonly used in prior log-related studies and are considered as following good logging practice [13], [17]–[20].

**Internal Validity.** Different hyper-parameters used in the neural networks might affect the effectiveness of the trained models. We follow the advanced practices from prior studies [27], [42], [58] to set the hyper-parameters for our deep learning framework. We conduct manual studies to investigate whether log level is implicitly or explicitly related to log message or the structural information of the logging statement. To avoid biases, two of the authors examine the data independently. For most of the cases, the authors reach an agreement. Any disagreement is discussed until a consensus is reached with a substantial-level agreement (Cohen's Kappa of 0.82 and 0.88 for logging location and log message, respectively) [26]. Involving third-party logging experts to verify our manual study results may further mitigate this threat.

**External Validity.** Our studied systems are all implemented in Java, the results and models may not be transferable to systems in other programming languages. We conducted our study on nine large-scale open source systems only. However, we selected the studied systems that are across various domains, different sizes, and different amount of logging statements in order to improve the representativeness of our studied systems. Future studies should validate the generalizability of our findings and the transferability of our models in systems that are implemented in other programming languages.

## VI. Related Work

**Studies on Logging Practices.** Chen et al. [59] and Yuan et al. [14] conducted quantitative studies on logging statements in large-scale open source C/C++ and Java systems, respectively. They found that logs are essential for debugging and maintenance purposes. Fu et al. [24] studied the logging practices in Microsoft software systems. They investigated what categories of code blocks (e.g., catch blocks) are logged. Li et al. [13] summarized the benefits and costs of logging through a qualitative study. Zhi et al. [60] studied how logging configurations are used in practice with respect to logging management, storage, and formatting. In this paper, we focus on studying the characteristics of log levels, specifically, their explicit or implicit relationship with the syntactic context or message of a logging statement. The findings of our study could complement prior studies in providing more comprehensive logging supports to developers.

**Improving Logging Practices.** Given the importance of logging, some studies try to help developers improve logging practices. Yuan et al. [7] proposed an approach that can automatically insert additional variables into logging statements to enhance the error diagnostic information. Zhu et al. [61] proposed an automated tool for suggesting logging locations. Li et al. [62], [63] proposed a deep learning framework for suggesting logging locations at the code block level. Liu et al. [27] proposed a deep learning framework to suggest the variables that should be recorded in logging statements. Chen et al. [17] found that developers commonly make some mistakes when writing logging statements (e.g., logging objects whose values may be null) and concluded five categories of logging anti-patterns from code changes. Li et al. [19], [64], [65] uncovered potential problems with logging statements that have the same text message and developed an automated tool to detect the problems. Hassani et al. [66] identified seven root-causes of the log-related issues from log-related bug reports and found that inappropriate log messages and missing log statements are the most common issues. Different from prior studies, we focus on suggesting log levels by using features extracted from the source code. We conduct a manual study on the characteristics of log levels and propose a deep learning based approach to provide automated suggestions.

## VII. Conclusion

Deciding proper log levels for logging statements is a challenging task. In this paper, we tackle the challenges in two steps. First, we conduct a manual study on the characteristics of log levels. We find that the syntactic context of logging statements and their messages, as well as the ordinal nature of log levels might be leveraged to help determine proper log levels. We then propose a deep-learning based approach to automatically suggest log levels for logging statements. Our approach ordinally encodes log levels and leverages the syntactic context information and the log message information of each logging statement to provide log level suggestions. Our approach outperforms the baseline approaches and are effective at suggesting log levels in both within-system and cross-system scenarios. Our results also highlight future research opportunities on improving logging decisions, for example, by leveraging the characteristics of different log levels to help distinguish similar log levels. Practitioners may also benefit from our findings to make better logging decisions.

## References

[1] B. Chen, J. Song, P. Xu, X. Hu, and Z. M. J. 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, ASE 2018*, 2018, pp. 305–316.

[2] T.-H. Chen, W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Cacheoptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, 2016, pp. 666–677.

[3] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16, 2016, pp. 102–111.

[4] J. Chen, W. Shang, A. E. Hassan, Y. Wang, and J. Lin, "An experience report of generating load tests using log-recovered workloads at varying granularities of user behaviour," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, 2019, pp. 669–681.

[5] Z. Li, "Towards providing automated supports to developers on writing logging statements," in *ICSE '20: 42nd International Conference on Software Engineering, Companion Volume*, 2020, pp. 198–201.

[6] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019*, 2019, pp. 683–694.

[7] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," in *ASPLOS '11: Proceedings of the 16th international conference on Architectural support for programming languages and operating systems*. ACM, 2011, pp. 3–14.

[8] 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 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 143–154.

[9] D. Schipper, M. F. Aniche, and A. van Deursen, "Tracing back log data to its log statement: from research to practice," in *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019*, 2019, pp. 545–549.

[10] K. Nagaraj, C. E. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI '12, 2012, pp. 353–366.

[11] M. Nagappan, K. Wu, and M. A. Vouk, "Efficiently extracting operational profiles from execution logs using suffix arrays," in *ISSRE'09: Proceedings of the 20th IEEE International Conference on Software Reliability Engineering*. IEEE Press, 2009, pp. 41–50.

[12] "Log4j," http://logging.apache.org/log4j/2.x/.

[13] H. Li, W. Shang, B. Adams, M. Sayagh, and A. E. Hassan, "A qualitative study of the benefits and costs of logging from developers' perspectives," *IEEE Transactions on Software Engineering*, pp. 1–17, 2020.

[14] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *ICSE 2012: Proceedings of the 2012 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2012, pp. 102–112.

[15] A. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Commun. ACM*, vol. 55, no. 2, pp. 55–61, Feb. 2012.

[16] 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.

[17] B. Chen and Z. M. J. Jiang, "Characterizing and detecting anti-patterns in the logging code," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17, 2017, pp. 71–81.

[18] ——, "Extracting and studying the logging-code-issue- introducing changes in java-based large-scale open source software systems," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2285–2322, Aug 2019.

[19] Z. Li, T. P. Chen, J. Yang, and W. Shang, "DLFinder: characterizing and detecting duplicate logging code smells," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*, 2019, pp. 152–163.

[20] H. Li, T.-H. P. Chen, W. Shang, and A. E. Hassan, "Studying software logging using topic models," *Empirical Software Engineering*, Jan 2018.

[21] "Simple logging facade for java (slf4j)," http://www.slf4j.org/faq.html, last checked Aug. 2020.

[22] S. Boslaugh and P. Watters, *Statistics in a Nutshell: A Desktop Quick Reference*, ser. In a Nutshell (O'Reilly). O'Reilly Media, 2008.

[23] H. Pirzadeh, S. Shanian, A. Hamou-Lhadj, and A. Mehrabian, "The concept of stratified sampling of execution traces," in *The 19th IEEE International Conference on Program Comprehension, ICPC 2011*, 2011, pp. 225–226.

[24] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? an empirical study on logging practices in industry," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE-SEIP '14, 2014, pp. 24–33.

[25] H. Pinjia, Z. Chen, S. He, and M. R. Lyu, "Characterizing the natural language descriptions in software logging statements," in *Proceedings of the 33rd IEEE international conference on Automated software engineering*, 2018, pp. 1–11.

[26] J. Sim and C. C. Wright, "The kappa statistic in reliability studies: Use, interpretation, and sample size requirements," *Physical Therapy*, vol. 85, no. 3, pp. 257–268, March 2005.

[27] Z. Liu, X. Xia, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Which variables should i log?" *IEEE Transactions on Software Engineering*, 2019, early Access.

[28] T.-H. Chen, S. W. Thomas, and A. E. Hassan, "A survey on the use of topic models when mining software repositories," *Empirical Software Engineering*, vol. 21, no. 5, pp. 1843–1919, 2016.

[29] "Corpus of contemporary american english," https://www.english-corpora.org/coca/, last checked Aug. 2020.

[30] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[31] Y. Huang, X. Hu, N. Jia, X. Chen, Y. Xiong, and Z. Zheng, "Learning code context information to predict comment locations," *IEEE Trans. Reliability*, vol. 69, no. 1, pp. 88–105, 2020.

[32] B. Li, H. Liu, Z. Wang, Y. Jiang, T. Xiao, J. Zhu, T. Liu, and C. Li, "Does multi-encoder help? A case study on context-aware neural machine translation," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, 2020, pp. 3512–3518.

[33] J. Tiedemann and Y. Scherrer, "Neural machine translation with extended context," in *Proceedings of the Third Workshop on Discourse in Machine Translation, DiscoMT@EMNLP 2017, Copenhagen, Denmark, September 8, 2017*, 2017, pp. 82–92.

[34] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 2016, pp. 1287–1293.

[35] M. L. Vásquez, C. McMillan, D. Poshyvanyk, and M. Grechanik, "On using machine learning to automatically classify software applications into domain categories," *Empir. Softw. Eng.*, vol. 19, no. 3, pp. 582–618, 2014.

[36] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *1st International Conference on Learning Representations, ICLR 2013*, 2013.

[37] P. D. Turney and P. Pantel, "From frequency to meaning: Vector space models of semantics," *J. Artif. Intell. Res.*, vol. 37, pp. 141–188, 2010.

[38] X. Li, H. Jiang, Y. Kamei, and X. Chen, "Bridging semantic gaps between natural languages and apis with word embedding," *IEEE Transactions on Software Engineering*, 2020.

[39] T. Hoang, J. Lawall, Y. Tian, R. J. Oentaryo, and D. Lo, "Patchnet: Hierarchical deep learning-based stable patch identification for the linux kernel," *IEEE Transactions on Software Engineering*, 2019.

[40] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering, ICSE 2018*, 2018, pp. 933–944.

[41] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *CoRR*, vol. abs/1901.01808, 2019.

[42] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*, 2019, pp. 783–794.

[43] B. D. Q. Nghi, Y. Yu, and L. Jiang, "Bilateral dependency neural networks for cross-language algorithm classification," in *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019*, 2019, pp. 422–433.

[44] Y. Xu, L. Mou, G. Li, Y. Chen, H. Peng, and Z. Jin, "Classifying relations via long short term memory networks along shortest dependency paths," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015*, 2015, pp. 1785–1794.

[45] "Keras: The python deep learning library," https://keras.io/, last checked Aug. 2020.

[46] "gensim Word2vec embeddings," https://radimrehurek.com/gensim/models/word2vec.html, last checked Feb. 2020.

[47] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.

[48] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "Deepjit: an end-to-end deep learning framework for just-in-time defect prediction," in *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019*, 2019, pp. 34–45.

[49] T. Zhang, C. Gao, L. Ma, M. R. Lyu, and M. Kim, "An empirical study of common challenges in developing deep learning applications," in *30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019*, 2019, pp. 104–115.

[50] 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.

[51] P. McCullagh, "Regression models for ordinal data," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 42, no. 2, pp. 109–127, 1980.

[52] H. Ha and H. Zhang, "Deepperf: performance prediction for configurable software with deep sparse neural network," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*, 2019, pp. 1095–1106.

[53] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*, 2019, pp. 25–36.

[54] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and F. Li, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009)*, 2009, pp. 248–255.

[55] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" in *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, 2014, pp. 3320–3328. [Online]. Available: http://papers.nips.cc/paper/5347-how-transferable-are-features-in-deep-neural-networks

[56] S. Kabinna, C. Bezemer, W. Shang, M. D. Syer, and A. E. Hassan, "Examining the stability of logging statements," *Empir. Softw. Eng.*, vol. 23, no. 1, pp. 290–333, 2018.

[57] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, J. Chen, X. He, R. Yao, J. Lou, M. Chintalapati, F. Shen, and D. Zhang, "Robust log-based anomaly detection on unstable log data," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, 2019, pp. 807–817.

[58] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*, 2017, pp. 135–146.

[59] B. Chen and Z. M. (Jack) Jiang, "Characterizing logging practices in java-based open source software projects – a replication study in apache software foundation," *Empirical Software Engineering*, vol. 22, no. 1, pp. 330–374, Feb 2017.

[60] C. Zhi, J. Yin, S. Deng, M. Ye, M. Fu, and T. Xie, "An exploratory study of logging configuration practice in java," in *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*, 2019, pp. 459–469.

[61] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE '15, 2015, pp. 415–425.

[62] Z. Li, T. Chen, and W. Shang, "Where shall we log? studying and suggesting logging locations in code blocks," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*, 2020, pp. 361–372.

[63] Z. Li, "Studying and suggesting logging locations in code blocks," in *ICSE '20: 42nd International Conference on Software Engineering, Companion Volume*, 2020, pp. 125–127.

[64] ——, "Characterizing and detecting duplicate logging code smells," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019*, 2019, pp. 147–149.

[65] Z. Li, T. P. Chen, J. Yang, and W. Shang, "Studying duplicate logging statements and their relationships with code clones," *IEEE Transactions on Software Engineering*, pp. 1–19, 2021.

[66] M. Hassani, W. Shang, E. Shihab, and N. Tsantalis, "Studying and detecting log-related issues," *Empirical Software Engineering*, 2018.