Leveraging Code Clones and Natural Language Processing for Log Statement Prediction

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

***Index Terms*—software systems, software automation, logging statement, logging prediction, source code, natural language processing, NLP, deep learning**

I. Introduction

To gather feedback about computer systems' running state, it is a common practice for developers to insert logging statements inside the source code to have running programs' internal state and variables written to log files. This logging process enables developers and system administrators to analyze log files for a variety of purposes [1], such as anomaly and problem detection [2, 3], log message clustering [4, 5], system profile building [6], code quality assessment [7], and compression of log files [4, 8]. Additionally, the wealth of information in the logs has also generated significant industrial interest and thus has initiated the development of commercialized log processing platforms such as Splunk [9] and Elastic Stack [10].

Due to the free-form text format of log statements and lack of a general guideline, adding proper logging statements to the source code remains a manual, inconsistent, and error-prone task [11]. As such, methods to automate logging ***location*** and predict the ***details***, *i.e.*, the *'static text'* and verbosity level of the logging statement, are well sought after. For example, the log print statement (LPS): log.warn("Cannot find BPService for bpid=" + id), contains a textual part indicating the context of the log, *i.e.*, *description*, *"Cannot find BPService for bpid="*, a *variable* part, *'id'*, and a log *verbosity level*,*'warn'*, indicating the importance of the logging statement and how the level represents the state of the program [12].

For practical concerns such as I/O and development costs, the *quantity*, *location*, and *description* of logging statements should be decided efficiently [13]. Logging too little may result in missing important runtime information that can negatively impact the postmortem dependability analysis [14], and excessive logging can consume extra system resources at runtime and impair the system's performance as logging is an I/O intensive task [15, 16]. In addition, due to the current *ad hoc* logging practices, developers often make mistakes in log statements or even forget to insert a log statement at all [17, 18]. Therefore, prior studies have aimed to automate the logging process and predict whether a code snippet requires a logging statement by utilizing machine learning methods to *train* a model on a set of logged code snippets, and then *test* it on a new set of unlogged code snippets [19, 20] (supervised learning). A recent work [21] has shown similar code snippets are useful for log statement description (LSD) suggestions by evaluating their *BLEU* [22] and *ROUGE* [23] scores, similar to *Precision* and *Recall*, respectively. Thus, in our research, we specifically seek to utilize source code clones for log statement prediction and suggestion.

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

II. Motivating Example

Source code clones are exact or similar snippets of the code that exist among one or multiple source code projects [24]. There are four main classes of code clones [25]: Type-1, which is simply copy-pasting a code snippet, Type-2 and Type-3, which are clones that show syntax differences to some extent, and finally Type 4, which represents two code snippets that are syntactically very different but semantically equal, *e.g.*, iterative versus recursive implementations of *Fibonacci* series in Figure 1. In this research, we focus on **method-level code clones** and call the tuple (MD_i, MD_j) a *'clone pair'*. Figure 1 shows that the logging pattern in the original code, MD_i on Line 3 can be learned to suggest logging statements for its clone, MD_j, which is missing a logging statement.

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

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

Fig. 1: Example for log prediction with code clones.

Practical Scenario. To illustrate how our approach will be useful for developers during the development cycle of the software, we provide the following practical scenario. We consider a possible employment of our research as a recommender tool, which can be integrated as a plugin to code development environments, *i.e.*, IDE software. Alex is a developer working on a large-scale software system and has previously developed method MD_i in the code base. At a later time, Dave, Alex's colleague, is implementing MD_j. Our automated log suggestion[1] approach can predict that if this new code snippet, MD_j, requires a logging statement by finding its clone, MD_i, in the code base. Then, the tool can suggest Dave, just in time, to add a log statement based on the prediction outcome.

III. Related Work

Prior work has tackled the automation of log statements with various approaches. Yuan *et al.* [14] proposed *ErrorLog*, a tool to report error handling code, *i.e.*, *error logging*, such as *catch clauses*, which are not logged and to improve the code quality and help with failure diagnosis by adding a log statement. Zhao *et al.* [15] introduced *Log20*, a performance-aware tool to inject new logging statements to the source code to disambiguate execution paths. *Log20* introduces a logging mechanism that does not consider developers' logging habits or concerns. Moreover, it does not provide suggestions for *logging descriptions*. Zhu *et al.* [20] proposed *LogAdvisor*, a learning-based framework, for automated logging prediction which aims to learn the frequently occurring logging practices automatically. Their method learns logging practices from existing code repositories for *exception* and *function return-value check* blocks by looking for textual and structural features within these code blocks with logging statements. Jia *et al.* [13] proposed an intention-aware log automation tool called *SmartLog*, which uses an *Intention Description Model* to explore the intention of existing logs, and Zhenhao et al. [26] categorized six block-level logging locations.

Recently, Li *et al.* [18] showed duplicate logging statements that are the outcome of shallow copy-pasting result in log-related anti-patterns (*i.e.*, issues). Although their research has a negative connotation towards copy-pasted logging statements from code clones, it simultaneously shows the potential of code clones as a starting point for automated log suggestion and improvement. In other words, by automating and enhancing the log statements in the clone pairs, we can expedite the development process and avoid shallow copy-pasting that developers tend to do. Additionally, by automation, we reduce the risk of irregular and *ad hoc* developers' logging practices, *e.g.*, forgetting to log in the first place.

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

IV. Research Approach

Based on the findings of He *et al.*[21] in logging description prediction based on *edit distance* [27], we hypothesize that similar code snippets, *i.e.*, code clones, follow similar logging patterns which can be utilized for log statement location and description prediction. Formally speaking, assuming set CC_{MD_i} is the set of all code clones of Method Definition MD_i, if MD_i has a log print statement (LPS), then its clones also have LPSs:

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

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

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

Our research design comprises a preliminary data collection phase, Stage 0, and is followed by four stages, Stages I-IV, to address RO1-RO4, as illustrated in Figure 2. In the following, we provide the details of our methodology and current results for each RO.

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

Motivation. To enable code clones for log suggestion, we first require to compare their characteristics and show if clone pairs follow similar logging patterns. ***Approach.*** For this purpose, we select seven large-scale open-source Java projects, *i.e.*, *Apache Hadoop, Zookeeper, CloudStack, HBase, Hive, Camel, and ActiveMQ*, based on the prior logging research [11, 21]. These projects are well-logged, stable, and well-used in the software engineering community, and also they enable us to compare our results with prior work, accordingly. We extract methods with logging statements and then find their clones. ***Evaluation.*** We evaluate the existence of log statements, their verbosity levels, and clone types.

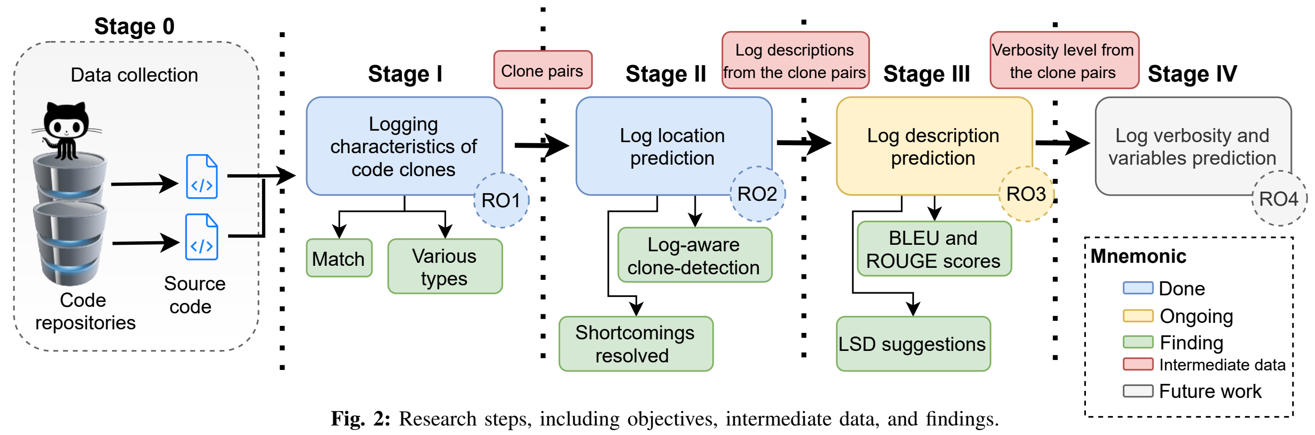

Fig. 2: Research steps, including objectives, intermediate data, and findings.

Results. The results show the majority of method clone pairs are consistent in their logging statements and their log verbosity levels also match to a high degree. Additionally, we find that the majority (in the range of 78% to 90%) of code clones are of Types 3 and 4, while the code pairs are matching in the existence of a logging statement. This observation signifies the effectiveness of code clones in suggesting the location of log statements in methods. In other words, although two snippets of clone pairs are syntactically different to a high degree, they still follow similar logging patterns.

*B. **RO2:** Propose an approach to utilize code clones for log statement location prediction.*

Motivation. Findings from RO1 show matching logging statements between clone pairs and motivate enablement of logging suggestions with code clones. The automated suggestion approach can help developers in making logging decisions and improve logging practices. ***Approach.*** We initially observe and resolve two shortcomings of general-purpose clone detectors to make them more suitable for log prediction and reduce false positive and false negative cases. We then utilize the clone pairs for suggesting logging statements for the methods which are missing an LPS by finding their clone pairs with a logging statement (Stage II in Figure 2). ***Evaluation.*** We evaluate the performance of our approach by measuring *Precision*, *Recall*, *F-Measure*, and *Balanced Accuracy* (BA) on the set of the seven selected projects. ***Results.*** Considering the average of BA values, our log-aware clone detection approach, LACCP, brings 15.60% improvement over Oreo [28] across the experimented projects. With the higher accuracy that LACCP brings, it enables us to provide more accurate clone-based log statement suggestions.

*C. **RO3:** Provide logging description suggestions based on code clones and NLP models.*

Motivation. Based on the experiment results for predicting the location of logging statements in RO2 and the additional available context from the clone pairs, *i.e.*, the logging statement description available from the original method, MD_i, we notice it is a valuable research effort to explore whether it is also possible to predict the logging statements' *description* automatically. With satisfactory performance, an automated tool that can predict the description of logging statements will be a great aid, as it can expedite the software development process and improve the quality of logging descriptions. ***Approach.*** We base our method on the assumption that clone pairs tend to have similar logging statement descriptions. This assumption comes from the observations in predicting log statements for clone pairs. As logging descriptions explain the source code surrounding them, it is intuitive for similar code snippets to have comparable logging descriptions. Based on this assumption, we propose a deep learning-based method that combines code clones with NLP learning approaches (NLP CC'd). In particular, to generate the LSD for a logging statement in MD_j, we extract its corresponding code snippet and leverage LACCP to locate its clone pairs. Laterally, the NLP model provides next word suggestions for the LSDs from the knowledge base available in the training set for each project. ***Evaluation.*** To measure the accuracy of our method in suggesting the log description, we utilize BLEU [22] and ROUGE [23] scores. These scores are well-established for validating the usefulness of an auto-generated text in prior software engineering and machine learning research, such as comment and code suggestion [29] and description prediction [21]. ***Results.*** We experiment on seven open-source Java systems, and our analysis shows that by utilizing log-aware clone detection and NLP, our hybrid model, (*NLP CC'd*), achieves 40.86% higher performance on BLEU and ROUGE scores for predicting LSDs when compared to the prior research [21], and achieves 6.41% improvement over the No-NLP version.

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

Motivation. Besides log statement location and its LSD, prediction of other details of log statements such as *Log verbosity level (LVL)* and its *variables (VAR)* are useful research efforts and the focus of prior research [30, 31, 32], as they further help the developers in more systematic logging and resolve suboptimal choices of log levels and variables [32]. ***Approach.*** Log-aware clone detection, LACCP, is reasonably extendable to predict LVL and VAR alongside the LSD suggestion. Since we have access to the source code of the method that we are predicting the logging statement for and its clone pair code snippet, a reasonable starting point is to suggest the same LVL as of its clone pair, and then augment it with additional learning approaches such as [30, 31] for more sophisticated LVL prediction. For VAR prediction, our approach can be augmented with deep learning [33] and static analysis of the code snippet under consideration [34] to include log variables suggestions along with the predicted LSD. ***Preliminary evaluation and results.*** Our preliminary analysis for the evaluated projects shows that code clones match in their verbosity levels in the range of (92, 97)%, which confirms that using the verbosity level of the clone pair, MD_i, is a good starting point for log verbosity level suggestions for MD_j. We are pursuing RO4 as our future work and will provide additional results and findings subsequently.

V. Discussion

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

Method-level log prediction rationale. Although clone detection (and subsequently, log statement prediction) can be performed in different granularity levels, such as files, classes, methods, and code blocks, however, method-level clones appear to be the most favorable points of re-factoring for all clone types [35]. We emphasize that our approach also includes all of the logging statements which are nested inside more preliminary code blocks within method definitions, *viz.*, logging statements nested inside code blocks, such as *if-else* and *try-catch*.

Comparison. Orthogonal to our research, prior efforts such as [20], [13], and [26] have proposed learning approaches for logging statements' *location* prediction, *i.e.*, *where to log*. The approaches in [20], [13] are focused on error logging statements (ELS), *e.g.*, log statements in *catch clauses*, and are implemented and evaluated on C# projects. Li *et al.* [26] provide log location suggestions by classifying the logged locations into six code-block categories. Different from these works, our approach does not distinguish between error and normal logging statements, is evaluated on open-source Java projects, and leverages logging statement suggestions at method-level by observing logging patterns in similar code snippets, *i.e.*, clone pairs.

Significance. Prior approaches [20, 26] rely on extracting features and training a learning model on logged and unlogged code snippets. Thus, they can predict if a new unlogged code snippet needs a logging statement by mapping its features to the learned ones. Although these methods initially appear similar to our approach in extracting log-aware features from code snippets, we believe our approach has an edge over the prior work. Because we also have access to the clone pair of the code under development, *i.e.*, MD_i in (MD_i, MD_j), this enables us to obtain and leverage the additional data from MD_i to predict other aspects of log statements, *e.g.*, LSD, which the prior work is unable to do. The significance of our approach becomes apparent in LSD prediction as we utilize the LSD of the clone pair as a starting point for suggesting the LSD of the new code snippet. Thus, our approach not only complements the prior work in providing logging suggestions for developers as they develop new code snippets, but it also has an edge over them by providing additional context for further prediction of LPS details, such as the LSD and the log's verbosity level.

VI. Summary of Contributions

The contributions that become available as the outcomes of our research are as follows: ❶ In RO1, we perform an experimental study on logging characteristics of code clones and show the potential for utilizing clone pairs for logging suggestions. ❷ In RO2, we introduce a log-aware clone detection tool (*LACCP*) for log statements' *'location'* prediction, and resolve two clone detection shortcomings for log prediction and provide experimentation on seven projects and compare it with general-purpose state-of-the-art clone detector, Oreo [28]. ❸ In RO3, we initially show the natural characteristics of software logs and that enables us to utilize our findings for the application of NLP for LSD prediction. We then propose a deep-learning NLP approach, *NLP CC'd*, to work in collaboration with *LACCP* to automatically suggest log statements' descriptions. We calculate the BLEU and ROUGE scores for our auto-generated log statements' *descriptions* by considering different sequences of LSD tokens, and compare our performance with the prior work [21]. ❹ Finally, as future work in RO4, we investigate the log verbosity level and variables prediction based on the information available through code clone pairs.

"One paragraph containing list of publications removed to preserve anonymity."

VII. Conclusions and Future Work

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

References

[1] C. Bertero, M. Roy, C. Sauvanaud, and G. Trédan, "Experience report: Log mining using natural language processing and application to anomaly detection," in *28th International Symposium on Software Reliability Engineering (ISSRE 2017)*, 2017, p. 10p.

[2] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 117–132.

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

[4] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Clustering event logs using iterative partitioning," in *Proceedings of the 15th ACM*

SIGKDD International Conference on Knowledge Discovery and Data Mining, ser. KDD '09. New York, NY, USA: ACM, 2009, pp. 1255–1264. [Online]. Available: http://doi.acm.org/10.1145/1557019.1557154

[5] R. Vaarandi and M. Pihelgas, "Logcluster-A data clustering and pattern mining algorithm for event logs," in *Network and Service Management (CNSM), 2015 11th International Conference on*. IEEE, 2015, pp. 1–7.

[6] A. Hassan, D. Martin, P. Flora, P. Mansfield, and D. Dietz, "An industrial case study of customizing operational profiles using log compression," in *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 2008, pp. 713–723.

[7] 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.

[8] L. Tang, T. Li, and C.-S. Perng, "Logsig: Generating system events from raw textual logs," in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, ser. CIKM '11. New York, NY, USA: ACM, 2011, pp. 785–794. [Online]. Available: http://doi.acm.org/10.1145/2063576.2063690

[9] "The Data-to-Everything Platform Built for the Cloud," https://www.splunk.com/, 2021.

[10] "Elasticsearch, the heart of the Elastic Stack," https://www.elastic.co/elasticsearch/, 2021.

[11] B. Chen and Z. M. J. 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, 2017.

[12] "The Apache Software Foundation. logging services project," http://logging.apache.org/.

[13] Z. Jia, S. Li, X. Liu, X. Liao, and Y. Liu, "Smartlog: Place error log statement by deep understanding of log intention," in *Software Analysis, Evolution and Reengineering (SANER), 2018 IEEE 25th International Conference on*. IEEE, 2018, pp. 61–71.

[14] 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 *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 293–306.

[15] 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*. ACM, 2017, pp. 565–581.

[16] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A cost-aware logging mechanism for performance diagnosis," in *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, 2015, pp. 139–150.

[17] M. Hassani, W. Shang, E. Shihab, and N. Tsantalis, "Studying and detecting log-related issues," *Empirical Software Engineering*, vol. 23, no. 6, pp. 3248–3280, 2018.

[18] Z. Li, T.-H. P. Chen, J. Yang, and W. Shang, "Studying duplicate logging statements and their relationships with code clones," *IEEE Transactions on Software Engineering*, 2021.

[19] 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 *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 24–33.

[20] 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-Volume 1*. IEEE Press, 2015, pp. 415–425.

[21] 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*. ACM, 2018, pp. 178–189.

[22] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 2002, pp. 311–318.

[23] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," *Text Summarization Branches Out*, 2004.

[24] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: scaling code clone detection to big-code," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 1157–1168.

[25] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

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

[27] E. S. Ristad and P. N. Yianilos, "Learning string-edit distance," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 5, pp. 522–532, 1998.

[28] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: Detection of clones in the twilight zone," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 354–365.

[29] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[30] 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, 2017.

[31] H. Anu, J. Chen, W. Shi, J. Hou, B. Liang, and B. Qin, "An approach to recommendation of verbosity log levels based on logging intention," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 125–134.

[32] Z. Li, H. Li, T.-H. P. Chen, and W. Shang, "Deeplv: Suggesting log levels using ordinal based neural networks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1461–1472.

[33] 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.

[34] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, p. 4, 2012.

[35] E. Kodhai and S. Kanmani, "Method-level code clone detection through lwh (light weight hybrid) approach," *Journal of Software Engineering Research and Development*, vol. 2, no. 1, p. 12, 2014.