Check for updates

Received: 26 October 2023 | Revised: 11 May 2024 | Accepted: 21 May 2024

DOI: 10.1002/smr.2696

RESEARCH ARTICLE - EMPIRICAL

WILEY

Deep learning or classical machine learning? An empirical study on line-level software defect prediction

Yufei Zhou[1] | Xutong Liu[2] | Zhaoqiang Guo[3] | Yuming Zhou[2] | Corey Zhang[4] | Junyan Qian[1]

[1]Key Lab of Education Blockchain and Intelligent Technology, Ministry of Education, Guangxi Normal University, Guilin, Guangxi, China

[2]State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China

[3]Huawei Technologies Co., Ltd, Hangzhou, Zhejiang, China

[4]Eastlake high school, Washington, USA

Correspondence
Junyan Qian, Key Lab of Education Blockchain and Intelligent Technology, Ministry of Education, Guangxi Normal University, Guilin, China.
Email: qjy2000@gmail.com

Funding information
National Natural Science Foundation of China, Grant/Award Numbers: 62162004, 62172205, U21A20474; Guangxi Natural Science Foundation, Grant/Award Numbers: 2024GXNSFBA010248, 2018GXNSFDA138003; Basic Ability Enhancement Program for Young and Middle-aged Teachers of Guangxi, Grant/Award Number: 2023KY0057

Abstract

Background: Line-level software defect prediction (LL-SDP) serves as a valuable tool for developers to detect defective lines with minimal human effort. Recently, GLANCE was proposed as a readily implementable baseline for assessing the efficacy of newly proposed LL-SDP models.

Problem: While DeepLineDP, a cutting-edge LL-SDP model rooted in deep learning, has demonstrated state-of-the-art performance, it has not yet been compared against GLANCE.

Objective: We aim to empirically compare DeepLineDP with GLANCE to obtain a comprehensive understanding of how deep learning contributes to solving the LL-SDP challenge.

Method: We compare GLANCE against DeepLineDP to assess the extent to which DeepLineDP surpasses GLANCE in predicting defective files and identifying problematic lines. In order to obtain a reliable conclusion, we use the same dataset and performance metrics utilized by DeepLineDP.

Result: Our experimental findings indicate that DeepLineDP does not outperform GLANCE in LL-SDP. This suggests that the application of deep learning, in this context, does not yield the anticipated significant improvements.

Conclusion: This finding underscores the need for further research in deep learning-based LL-SDP to attain the state-of-the-art performance that remains elusive for less advanced techniques.

KEYWORDS
code features, deep learning, defect prediction, line-level

1 | INTRODUCTION

Line-level software defect prediction (LL-SDP) predicts whether a line of code is defective using dedicated prediction models.[1–5] Unlike traditional defect prediction models that operate at coarser granularities (such as file-level or function-level defect prediction[6–12]), LL-SDP operates at a finer level of granularity by focusing on individual lines of code. By assessing individual lines of code, it allows for the precise pinpointing of defective locations within a software system. This pinpoint accuracy is invaluable to developers, as it enables them to focus their attention and efforts on specific lines that require immediate attention, rather than dealing with entire files or functions.

Over the past few decades, significant efforts have been dedicated to the field of LL-SDP. Presently, this area of research can be categorized into three primary domains.[2] The first category involves the use of automatic static analysis tools to detect potentially problematic lines of

© 2024 John Wiley & Sons Ltd.

code.[13–15] If code lines conform to predefined bug patterns, they are flagged as potentially buggy. The second category employs natural language processing techniques to identify potentially buggy lines.[4,5] These approaches typically construct statistical language models, such as *n*-grams, to assess the "naturalness" or the suspicion score of source code. The underlying premise is that less "natural" code is more likely to harbor defects. The third category leverages model interpretation techniques to predict defective lines.[3] These approaches achieve LL-SDP by interpreting the contributions of tokens within the context of a coarse-grained defect prediction model (typically at the file-level). Consequently, lines that contain risky tokens are identified as potentially buggy.

The aforementioned three categories encompass a range of strategies employed within the field of LL-SDP. Each of these categories offers unique insights and approaches to address the challenge of identifying defective lines within software code. However, despite these advancements, there remains uncertainty about the extent of progress made in solving the LL-SDP problem. To shed light on this matter, Guo et al conducted an extensive experiment with the aim of evaluating the effectiveness of state-of-the-art (SOTA) LL-SDP approaches.[2] Specifically, they introduced a straightforward heuristic baseline solution called GLANCE, along with three implementations (GLANCE-MD, GLANCE-EA, and GLANCE-LR), as the baseline model. Their experimental findings revealed that GLANCE exhibited prediction performance comparable to or even superior to existing SOTA LL-SDP approaches. Consequently, they recommended the adoption of GLANCE as a readily implementable baseline in future studies to showcase the utility of newly proposed LL-SDP approaches.

Recently, Pornprasit et al made a significant contribution to the field of LL-SDP by introducing deep learning techniques.[1] Their novel approach, DeepLineDP, harnesses the power of deep learning to autonomously extract semantic properties from the surrounding tokens and lines, facilitating the identification of defective lines within software code. In their study, Pornprasit et al conducted experiments employing three SOTA LL-SDP approaches as the baselines: n-gram, ErrorProne, and random forest. This comprehensive analysis involved a case study encompassing 32 releases from nine projects. Remarkably, their findings revealed that DeepLineDP exhibited remarkable cost-effectiveness, outperforming other LL-SDP approaches by a substantial margin—specifically, it demonstrated a cost-effectiveness improvement ranging from 47% to an impressive 250%. While DeepLineDP has unquestionably demonstrated state-of-the-art performance in LL-SDP, an intriguing aspect remains to be explored: a direct comparison with the GLANCE approach. This comparison represents an intriguing avenue for further research in the realm of LL-SDP, with the potential to shed light on the strengths and weaknesses of these two distinct approaches. Such insights could ultimately contribute to the refinement and advancement of LL-SDP methodologies, further enhancing our ability to identify and mitigate defects in software code.

In this study, we conduct a comprehensive empirical comparison between DeepLineDP and GLANCE, with the aim of gaining a deep understanding of how deep learning contributes to addressing the challenges of LL-SDP. To this end, we employ a rigorous methodology to conduct the experiment. To begin, we utilize the replication package of DeepLineDP, ensuring a faithful reproduction of the original experiment. Simultaneously, we employ the shared codebase of GLANCE, specifically tailored to the same dataset initially used by DeepLineDP. This dual approach allows us to explore the extent to which DeepLineDP surpasses GLANCE in its predictive accuracy for identifying defective lines in software code. To maintain consistency and comparability with the original DeepLineDP study, we adopt the same set of performance metrics. This meticulous setup ensures that we utilize the original implementations of both DeepLineDP and GLANCE, thereby eliminating potential sources of bias and ensuring the objectivity of our conclusions. The results of our rigorous experimentation reveal that GLANCE is comparable or even superior to DeepLineDP, a finding that challenges the prevailing belief that the application of deep learning would yield significant improvements in LL-SDP. This finding underscores the pressing need for further exploration in the domain of deep learning-based LL-SDP. Achieving the elusive state-of-the-art performance, which has thus far remained beyond the reach of less advanced techniques, necessitates a deeper and more nuanced understanding of the underlying mechanisms and challenges in this field. In essence, our study emphasizes the importance of continued research efforts to unlock the full potential of deep learning in the context of LL-SDP.

In summary, we make the following contributions to this study:

- **Empirical Comparison of DeepLineDP and GLANCE**. DeepLineDP and GLANCE, both published in 2023, employ different methodologies, with one relying on manually designed features, and the other utilizing features automatically generated through deep learning. However, the comparative superiority of these approaches remains unexplored. Our study fills this gap with an empirical comparison between DeepLineDP and GLANCE.
- **Challenging prevailing assumptions**. Our finding challenges the prevailing assumption that the application of deep learning would inherently lead to significant improvements in LL-SDP. By demonstrating that DeepLineDP does not outperform GLANCE, we contribute to a nuanced understanding of the impact of deep learning in this field.
- **Highlighting the need for further research**. Our study emphasizes the need for continued research in deep learning-based LL-SDP to achieve elusive state-of-the-art performance through a deeper understanding of underlying challenges.

The rest of this paper is organized as follows: Section 2 offers an introduction to contemporary line-level defect prediction models. In Section 3, we present an illustrative motivational example that inspired this study. Our study's design is outlined in Section 4, while Section 5 presents the experimental results. Factors influencing our findings are discussed in Section 6, followed by the implications of our findings in Section 7

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

and an analysis of potential validity threats in Section 8. Section 9 introduces related work, and Section 10 concludes the paper while also outlining directions for future research.

2 | STATE-OF-THE-ART LINE-LEVEL DEFECT PREDICTION MODELS

In this section, we introduce two SOTA LL-SDP models: GLANCE and DeepLineDP. GLANCE serves as our baseline model, leveraging straightforward code characteristics, while DeepLineDP represents the forefront of LL-SDP models, harnessing the power of deep learning techniques.

2.1 | GLANCE

After analyzing defective code across multiple projects, Guo et al made the following observations[2]: (1) defective code lines tend to exhibit high complexity, as measured by both their code size and the number of function calls; and (2) many defective code lines are associated with control structures such as "for," "if," and "while" statements. Based on these observations, Guo et al hypothesized that two types of code lines would be more likely to contain defects:

- Control lines. These are lines containing at least one control element, which can be categorized into three types: branch control, loop control, and jump control, depending on their effects.
- Complex lines. These lines have intricate structures, which can be measured along two dimensions: the number of function calls and the number of code tokens.

Building upon these insights, Guo et al introduced a straightforward heuristic baseline solution called GLANCE (short for "aiming at Control-And ComplEx-statements"). The name reflects the idea that one can quickly identify potentially buggy lines by examining control structures and complexity metrics in code lines. GLANCE operates as a two-stage framework. In the first stage, a file-level classifier is constructed to predict defective files within a target project. In the second stage, they calculate the defect-proneness score for a given code line l as follows:

$$DefectPronessScore(l) = NT(l) \times [NFC(l) + 1]$$

where NT measures the complexity of a code line in terms of the number of code tokens, and NFC measures complexity in terms of the number of function calls. GLANCE then arranges all code lines within a file in descending order of their corresponding *DefectPronessScore*, moving lines containing control elements to the front of the ranking list. If code lines have tied ranks, they will be prioritized based on their respective orders (i.e., line number) in the file. This strategy simulates developers' typical practice of reviewing code from the beginning to the end.

In Guo et al's study,[2] they examined three distinct GLANCE implementations, each paired with different file-level classifiers: GLANCE-MD (utilizing the ManualDown classifier), GLANCE-EA (employing the Effort Aware ManualDown classifier), and GLANCE-LR (utilizing the logistic regression classifier). In contrast to GLANCE-LR, both GLANCE-MD and GLANCE-EA are simple unsupervised classifiers. These classifiers operate based on the code size of modules, measured by the source lines of code in the target project, independent of the source project. Their experimental findings showed that GLANCE-LR outperformed both GLANCE-MD and GLANCE-EA. Notably, they reported that GLANCE-LR exhibited prediction performance on par with or even surpassing existing state-of-the-art LL-SDP approaches, including those based on natural language processing, model interpretation techniques, and static analysis tools. Consequently, they recommended the adoption of GLANCE-LR as an easily implementable baseline for LL-SDP in future studies.

2.2 | DeepLineDP

DeepLineDP is meticulously designed to detect defective code lines by harnessing the power of bidirectional GRU units, which learn from adjacent tokens and lines using a hierarchical attention network.[1] At its core, it maintains the hierarchical structure of source code, wherein tokens form lines and lines form files. DeepLineDP uses a two-layer attention network, encompassing a token layer and a line layer. This network incorporates four essential components: a token encoder, a token-level attention layer, a line encoder, and a line-level attention layer.

In the token-level processing, DeepLineDP generates a vector representation for each token using Word2Vec.[16] The token encoder leverages a bidirectional GRU[17] to amalgamate information from both the preceding and succeeding contexts of the token, resulting in an annotation that encapsulates its contextual information. Subsequently, DeepLineDP employs the attention mechanism to accentuate tokens pivotal to the semantics of the source code lines and aggregates these informative token representations to construct a line vector. In a parallel manner, for each line,

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

the line encoder employs a bidirectional GRU to derive an annotation encoding its contextual information. Afterward, DeepLineDP utilizes the attention mechanism to spotlight lines that are crucial to the semantics of the source code file and consolidates the representations of these informative lines into a file vector. This file vector is then fed into a one-layer Multi-Layer Perceptron (MLP) to calculate the probability of the file being defective.

In the final step, the token-level attention layer is employed to identify faulty lines based on the most critical tokens contributing to the prediction of defective files. To discern these faulty lines, DeepLineDP extracts the attention score for each code token within the defective file. This attention score serves as a proxy for indicating the riskiness of code tokens. To accomplish this, DeepLineDP ranks the attention scores obtained from the token attention layer in descending order and selects the top-k tokens with the highest attention scores. Subsequently, DeepLineDP computes the line-level risk score as the sum of the risk scores for any tokens appearing in the top-k. Finally, DeepLineDP generates a ranking of the most risky lines based on the assigned risk scores for each line. In their study, the value of k is set to 1,500.

In an extensive case study involving 32 releases of 9 software projects, Pornprasit et al reported remarkable results for DeepLineDP.[1] It exhibited a 17%-37% increase in accuracy compared to four file-level defect prediction approaches: Bag of Words (BoW), Deep Belief Network (DBN), Convolutional Neural Network (CNN), and Bidirectional Long Short-Term Memory (BiLSTM). Moreover, it demonstrated a substantial 47%-250% increase in cost-effectiveness when compared to three line-level defect prediction approaches: n-gram, ErrorProne, and random forest. These findings underscore the critical importance of considering surrounding tokens and lines to pinpoint the precise locations of defective files, namely, defective lines.

3 | MOTIVATING EXAMPLE

In 2023, both GLANCE and DeepLineDP were introduced, representing distinct methodologies. GLANCE relies on conventional manually designed features, while DeepLineDP is founded on automatically generated features utilizing cutting-edge deep learning techniques. This contrast prompts a crucial query: how does DeepLineDP compare with GLANCE? Addressing this question is pivotal in illuminating the actual progress in current line-level defect prediction.

To investigate this problem, we evaluated the performance of GLANCE using the identical datasets employed in the original DeepLineDP study. Table 1 presents the results of the line-level defect prediction for both GLANCE-LR and DeepLineDP, applied to the file PreAddPartitionEvent.java in the hive-0.12.0 target project. The first column enumerates the line number for each line, where lines 1-17 and 33-35 are comments, and lines 18, 20, 23, 25, 27, and 32 are blank. Among the remaining 13 non-comment and non-blank code lines, 3 lines (31, 38, and 39) consist solely of "}". Based on DeepLineDP and GLANCE, only the following 10 lines are deemed defect-prone: 19, 21, 22, 24, 26, 28, 29, 30, 36, and 37 (highlighted in yellow background in the first column). The second column provides the code for PreAddPartitionEvent.java. According to the line-level defect oracle information provided by,[1] the following five lines are indeed defective: 26, 28, 30, 36, and 37 (highlighted in amber background). The third column lists the metrics NT (Number of Tokens), NFC (Number of Function Calls), and CE (Control Elements) utilized to calculate the line score, along with the resulting rank under GLANCE. The last column displays the line score (i.e., line attention) predicted by DeepLineDP, along with the corresponding rank for each line.

From the data presented in Table 1, the following observations can be made. Under GLANCE-LR, the defective lines 37 and 28 occupy the top two positions in the rankings for line-level defect prediction. This results in the following performance metrics: Recall@Top20%LOC = 0.4, Effort@Top20%Recall = 0.1, and IFA = 0. These metrics collectively indicate that under the GLANCE-LR model, software developers can achieve the following: (1) identify 40% of actual defective lines by inspecting only the top 20% of lines of code (LOC), (2) uncover the top 20% of actual defective lines by scrutinizing a mere 10% of the code lines, and (3) detect the initial defective line without the need to examine any preceding lines. Contrastingly, in the case of DeepLineDP, all five defective lines receive low rankings (with line 26 being the highest at rank 4). As a result, the performance metrics are as follows: Recall@Top20%LOC = 0, Effort@Top20%Recall = 0.4, and IFA = 0. From these results, it is evident that GLANCE-LR outperforms DeepLineDP in line-level defect prediction for PreAddPartitionEvent.java. This suggests that, at least in this particular example, DeepLineDP does not exhibit an advantage over GLANCE-LR, especially considering that DeepLineDP is computationally intensive while GLANCE-LR is computationally efficient.

Intuitively, given the use of state-of-the-art deep learning techniques capable of generating powerful semantic features, one might expect DeepLineDP to outperform GLANCE, which relies on simple syntactical features. However, the example above indicates that this may not be the case. To arrive at a dependable conclusion, our next step involves conducting a comprehensive experimental comparison of the line-level defect prediction capabilities between DeepLineDP and GLANCE.

4 | STUDY DESIGN

In this section, we will outline our study design, encompassing the research questions, datasets, analysis methodology, evaluation metrics, and statistical analysis method.

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 1 The line-level defect prediction performance when applying GLANCE-LR and DeepLineDP to "metastore/src/java/org/apache/hadoop/hive/metastore/events/PreAddPartitionEvent.java" in the target project hive-0.12.0.

		GLANCE-LR					DeepLineDP	
Line	PreAddPartitionEvent.java	NT	NFC	Score	CE	Rank	Attention score	Rank
1	/**							
2	* Licensed to the Apache Software Foundation (ASF) under one							
	...							
17	*/							
18								
19	package org.apache.hadoop.hive.metastore. events;	7	0	7	0	6	0.99994306	3
20								
21	import org.apache.hadoop.hive.metastore. HiveMetaStore.HMSHandler;	8	0	8	0	3	0.99996625	1
22	import org.apache.hadoop.hive.metastore.api. Partition;	8	0	8	0	4	0.99996621	2
23								
24	public class PreAddPartitionEvent extends PreEventContext {	5	0	5	0	8	0.99721108	5
25								
26	private final Partition partition;	4	0	4	0	9	0.99948066	4
27								
28	public PreAddPartitionEvent (Partition partition, HMSHandler handler) {	6	1	12	0	2	0.13964098	6
29	super (PreEventType.ADD_PARTITION, handler);	4	1	8	0	5	0.07722895	8
30	this. partition = partition;	3	0	3	0	10	0.08214881	7
31	}							
32								
33	/**							
34	* @return the partition							
35	*/							
36	public Partition getPartition() {	3	1	6	0	7	0.06477406	9
37	return partition;	2	0	2	1	1	0.04723226	10
38	}							
39	}							
Recall@Top20%LOC		0.4 (better)					0	
Effort@Top20%Recall		0.1 (better)					0.4	
IFA		0 (better)					3	

4.1 | Research questions

To obtain a comprehensive understanding of how deep learning contributes to solving the LL-SDP challenge, we investigate the following research questions:

RQ1 (Effectiveness in predicting defective files): To what extent does DeepLineDP outperform GLANCE in its ability to predict defective files?

The aim of RQ1 is to assess DeepLineDP's efficacy in terms of file-level classification performance as compared to GLANCE. GLANCE, being a baseline approach with a simplistic structure, is pitted against DeepLineDP, which boasts a more intricate architecture. Consequently, our primary inquiry is whether DeepLineDP surpasses GLANCE in its ability to predict defective files with superior classification performance. The resolution of this research question holds the key to illuminating the utility of DeepLineDP and affording us a genuine understanding of its performance in the realm of file-level defect prediction.

RQ2 (Cost-effectiveness in locating defective lines): To what extent does DeepLineDP outperform GLANCE in its ability to locate defective lines?

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

The aim of RQ2 is to evaluate DeepLineDP's effectiveness in terms of line-level ranking performance relative to GLANCE. Our central investigation revolves around whether DeepLineDP outperforms GLANCE in terms of cost-effectiveness for identifying defective lines, specifically with superior ranking performance. The answer to this research question is pivotal in shedding light on the practical utility of DeepLineDP and providing us with a comprehensive understanding of its performance in the context of line-level defect prediction.

RQ3: (Characteristics of detected defective lines): Are there variations in the characteristics of detected defective lines, including defect categories, sizes, and locations, when comparing DeepLineDP with GLANCE?

The aim of RQ3 is to thoroughly examine potential disparities in the characteristics of identified defective lines. More precisely, this investigation aims to ascertain whether variations exist in defect categories, sizes, and locations among detected defective lines when comparing DeepLineDP with GLANCE. This inquiry contributes to a comprehensive understanding of how each approach may excel or differ in addressing specific aspects of defective lines, providing valuable insights for evaluating their respective strengths and weaknesses.

4.2 | Data sets

In our study, we utilize the same defect dataset employed in the study by Pornprasit et al.[1] This dataset comprises 9 popular open-source Java systems, encompassing a total of 32 different releases. An overview of this dataset is presented in Table 1. The first two columns provide the project's name and description, while the third and fourth columns offer file-level information, including the range of the number of files and the range of the proportion of buggy files across the subject releases, respectively. The fifth and sixth columns present line-level details for each project, encompassing the range of the number of lines of code and the range of the proportion of buggy lines across the subject releases. The final column lists the subject releases for each project. Notably, these projects under examination exhibit variability in application domains, size, and the proportion of faulty files. It is worth mentioning that this dataset has been employed in many previous research studies.[1,18,19]

4.3 | Analysis methodology

We conducted a comprehensive comparative analysis between DeepLineDP and GLANCE, using a cross-release evaluation methodology across multiple software projects. In this evaluation framework, we considered each software project, assuming it had *n* releases. To ensure a consistent basis for comparison, we followed a standardized procedure:

1. Release chronology ranking. We initially organized the releases of each project chronologically, based on their release dates. This step ensured that the evaluation would proceed systematically, from older to newer versions.
2. DeepLineDP evaluation. For DeepLineDP, we adopted a training-validation-testing scheme. We used release 1 as the training dataset and release 2 as the validation dataset to train the model. Subsequently, we tested the trained model on the subsequent releases (i.e., releases 3 through n). This evaluation setting mirrors the approach utilized in a prior study by Pornprasit et al.[1]
3. GLANCE evaluation. In contrast, GLANCE employed a distinct evaluation approach. For GLANCE, we employed a sliding window evaluation strategy. Specifically, for release *i*, it was utilized as the training dataset, while release $i+1$ served as the test dataset. We applied this strategy for releases 2 through n, adhering to the evaluation methodology established by Guo et al.[2]

It is worth noting that both DeepLineDP and GLANCE used the same test sets, ensuring a direct comparison of their performances. The primary distinction lay in the choice of training data, which varied for each technique. This systematic evaluation approach yielded a total of 14 unique train-test evaluation combinations, corresponding to various projects: 3 combinations for ActiveMQ, 2 combinations for Camel, 2 combinations for JRuby, 2 combinations for Lucene, 1 combination for Derby, 1 combination for Groovy, 1 combination for Hbase, 1 combination for Hive, and 1 combination for Wicket. Finally, to assess the effectiveness of each technique on a specific project, we individually evaluated the results for each combination and computed their average performance. This comprehensive analysis allowed us to estimate the overall performance of an LL-SDP approach across a diverse range of software projects.

For GLANCE and DeepLineDP, we conduct a manual analysis to identify potential differences in their recognition of true defective lines. This involves randomly selecting a statistically significant sample of true defective lines identified by each defect prediction model. Following established guidelines, our sample is determined with a 95% confidence level and a 5% margin of error.[20] The sample sizes for DeepLineDP, GLANCE-MD, GLANCE-EA, and GLANCE-LR are 371, 370, 365, and 366, respectively. These sizes are determined based on the differing total numbers of true defective lines identified by each defect prediction model within the dataset, which are 10,735, 10,397, 7,900, and 7,438,[*] respectively. Subsequently, we examine variations in the characteristics of the detected defective lines, such as defect category, size, and location. Independent inspections of each true defective line are carried out by two authors to determine its category. In case of any inconsistencies, the

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

authors engage in discussions to reach a consensus on the final category. To gauge the agreement level between the two authors when categorizing the same samples, we compute Cohen's Kappa coefficient.[21] Cohen's Kappa coefficients range from 0 to 1, with values closer to 1 indicating higher consistency between evaluators.

4.4 | Evaluation metrics

We employ the same performance metrics utilized in the study conducted by Pornprasit et al for our evaluation. As previously mentioned, LL-SDP involves two distinct stages: first, the prediction of defective files, and second, the localization of defective lines within the predicted defective files. The former stage is formulated as a classification task at the file level, while the latter is treated as a ranking task at the line level.

For the file-level classification, we utilize the following evaluation metrics:

- *AUC* (*Area Under the Curve*): AUC represents the area under the Receiver Operating Characteristic (ROC) curve, which plots the true positive rate against the false positive rate.[22] AUC values range from 0 to 1, where a value of 1 signifies perfect discrimination, and a value of 0.5 indicates random guessing.
- *Balanced Accuracy*: Balanced Accuracy computes the mean of the True Positive Rate (i.e., the ratio of correctly predicted defective files, TP/[TP + FN]) and the True Negative Rate (i.e., the ratio of correctly predicted clean files, TN/[TN + FP]). In this context, TP, TN, FP, and FN represent True Positive, True Negative, False Positive, and False Negative, respectively. A high balanced accuracy score signifies the capability of an approach to make accurate predictions for both defective and clean files.
- *MCC* (*Matthews Correlation Coefficient*): MCC measures the correlation coefficient between actual and predicted outcomes. According to,[23] an MCC value of 1 indicates a perfect prediction, and conversely, a lower value indicates a weaker prediction. Consequently, a higher MCC value corresponds to a more effective model.

These performance metrics provide a comprehensive assessment of the LL-SDP approach's performance in predicting defective files at the file level. It is crucial to emphasize that when dealing with classification tasks, Balanced Accuracy and MCC hold greater significance for practitioners compared to AUC. This preference arises from the practical reality that, in most real-world scenarios, only the instances predicted as potentially defective or belonging to the positive class are subjected to further testing or inspection. However, AUC measures the overall discriminative power of a classifier across a range of decision thresholds but does not consider the specific operational conditions of the task. In other words, it may not directly address the primary concerns of practitioners who are focused on isolating and dealing with potentially defective instances.

For the line-level ranking, we utilize the following evaluation metrics:

- *Recall@Top*20%*LOC*. This metric assesses the ability to precisely identify defective lines within the top 20% LOC within the entire release. A high value indicates that an LL-SDP approach effectively ranks a substantial number of genuine defective lines at the top, streamlining the process of identifying these defects with minimal effort. Conversely, a low value suggests that a noteworthy portion of clean lines is located within the top 20% of LOC, demanding developers to invest more effort in pinpointing defective lines.
- *Effort@Top*20%*Recall*. This metric quantifies the amount of effort, measured in LOC, needed to identify the 20% of actual defective lines within the entire release. A low value suggests that developers require only a minimal amount of effort to locate the top 20% of actual defective lines. Conversely, a high value indicates that developers need to invest a significant amount of effort to uncover the top 20% of actual defective lines.
- *IFA* (*Initial False Alarm*). This metric represents the number of lines that must be examined before identifying the first defective line.[24] As indicated by Parnin et al,[25] developers typically prefer to halt their inspection once they no longer encounter defective lines among the top recommended lines. Therefore, a lower IFA value signifies a more effective LL-SDP model, as it necessitates fewer line inspections to uncover the initial buggy line.

In summary, these line-level evaluation metrics offer a multi-dimensional assessment of the LL-SDP approach's proficiency in prioritizing defective lines within software releases.

4.5 | Statistical analysis

We employ the ScottKnott Effect Size Difference (ESD) test to meticulously categorize distributions into statistically distinct ranks, as highlighted in references [26, 27]. The ScottKnott ESD test delivers a ranking of these techniques, all the while guaranteeing two critical criteria:

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

- Minimized intra-rank variation. Within each rank, the ScottKnott ESD test ensures that the magnitude of the difference between the distributions is statistically negligible. This stringent criterion guarantees that the techniques within the same rank are indeed similar in their effects, reducing the likelihood of false distinctions.
- Highlighted inter-rank disparities. Conversely, between different ranks, the test brings to the forefront the magnitude of the differences in distributions, emphasizing their non-negligible nature. This emphasizes the significance of disparities between ranks, helping to identify the techniques that exhibit distinct and noteworthy effects.

By adhering to these two criteria, we achieve a comprehensive and rigorous categorization of techniques, allowing us to draw meaningful conclusions about their relative performance and characteristics. This approach empowers us to discern not only which techniques excel in certain aspects but also to gauge the significance of these differences in a statistically robust manner.

5 | EXPERIMENTAL RESULTS

In this section, we provide a comprehensive account of the experimental results obtained from our comparative analysis between GLANCE and DeepLineDP. Our focus is on assessing the performance of these two approaches in terms of their ability to predict defective files and their cost-effectiveness in pinpointing defective lines within the codebase.

5.1 | RQ1: effectiveness in predicting defective files

Figures 1, 2, and 3 report the ScottKnott ESD rankings in conjunction with the corresponding distribution plots for AUC, Balanced Accuracy, and MCC metrics, facilitating a thorough comparative assessment of GLANCE and DeepLineDP in each scenario. To deepen our comprehension, Table 2 furnishes an exhaustive breakdown of these performance metrics, delivering a detailed examination of the predictive capabilities of GLANCE and DeepLineDP across 14 target releases within a diverse range of nine open-source subject projects in identifying defective files.

From Figures 1, 2, 3, and Table 2, the following observations can be made:

- AUC. As depicted in Figure 1, the ScottKnott ESD test unveils a three-tier ranking structure: GLANCE-EA and GLANCE-MD occupy the top rank, DeepLineDP secures the second rank, and GLANCE-LR falls into the third rank. This ranking alignment is substantiated by the AUC findings in Table 2, where GLANCE-EA and GLANCE-MD exhibit the most substantial mean/median AUC, while GLANCE-LR registers the lowest mean/median AUC.
- Balanced Accuracy. As shown in Figure 2, the ScottKnott ESD test unveils a four-tier ranking structure. GLANCE-MD and GLANCE-EA emerge as the leaders in the top two tiers, while GLANCE-LR solidifies its position in the third tier. Meanwhile, DeepLineDP takes its place in the fourth tier. This hierarchical arrangement is corroborated by the Balanced Accuracy results presented in Table 2. Specifically, GLANCE-MD and GLANCE-EA exhibit the highest mean and median Balanced Accuracy values, underscoring their superior performance. In contrast, DeepLineDP records the lowest mean and median Balanced Accuracy scores, highlighting its comparatively lower accuracy.

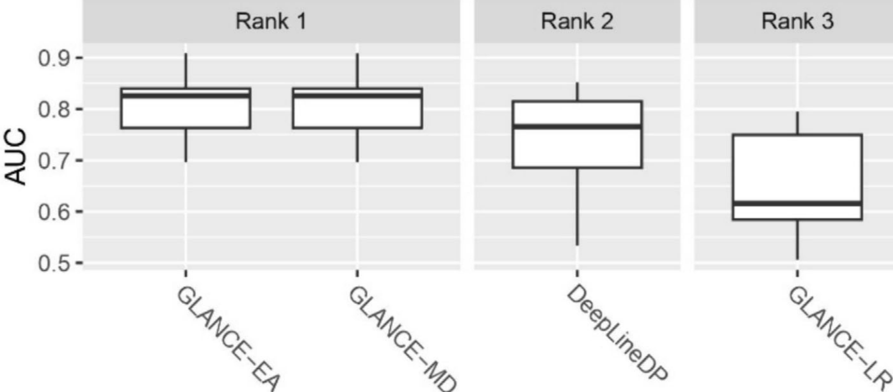


FIGURE 1 The ScottKnott ESD ranking on AUC: DeepLineDP vs. GLANCE.

Journal of Software: Evolution and Process — WILEY

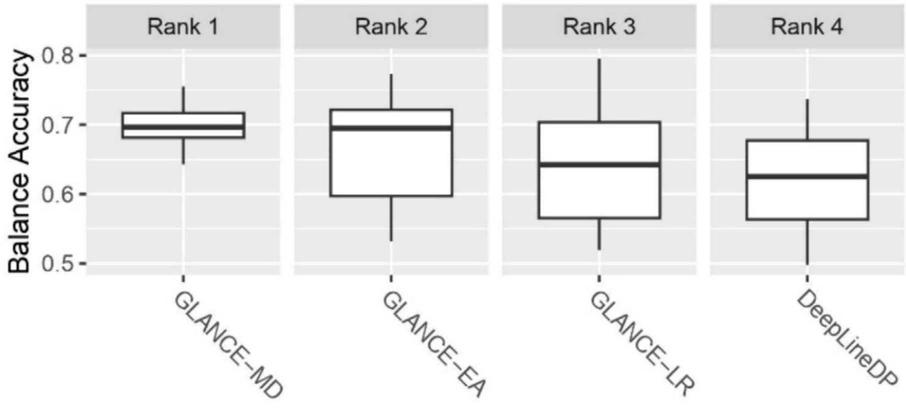


FIGURE 2 The ScottKnott ESD ranking on balanced accuracy: DeepLineDP vs. GLANCE.

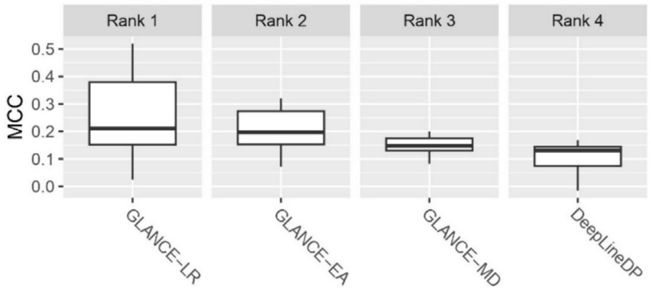


FIGURE 3 The ScottKnott ESD ranking on MCC: DeepLineDP vs. GLANCE.

TABLE 2 The detailed statistical information of all releases in each studied project.

		File-level		Line-level		
Project	**Description**	**#files**	**%buggy**	**#lines**	**%buggy**	**Releases**
ActiveMQ	Messaging and integration patterns	1884-3,420	2%-7%	142 k-299 k	0.08%-0.44%	5.0.0, 5.1.0, 5.2.0, 5.3.0, 5.8.0
Camel	Enterprise integration framework	1,515-8,846	2%-8%	75 k-485 k	0.09%-0.24%	1.4.0, 2.9.0, 2.10.0, 2.11.0
Derby	Relational database	1963-2,705	6%-28%	412 k-533 k	0.10%-0.63%	10.2.1.6, 10.3.1.4, 10.5.1.1
Groovy	Java-syntax-compatible OOP	757-884	2%-4%	74 k-93 k	0.10%-0.17%	1.5.7, 1.6.0.Beta_1, 1.6.0.Beta_2
HBase	Distributed scalable data store	1,059-1834	7%-11%	246 k-537 k	0.17%-1.02%	0.94.0, 0.95.0, 0.95.2
Hive	Data warehouse system for Hadoop	1,416-2,662	6%-19%	290 k-567 k	0.31%-2.90%	0.9.0, 0.10.0, 0.12.0
JRuby	Programming language for JVM	731-1,614	2%-13%	106 k-240 k	0.03%-0.09%	1.1, 1.4, 1.5, 1.7
Luence	Text search engine library	805-2,806	2%-8%	101 k-342 k	0.07%-0.39%	2.3.0, 2.9.0, 3.0.0, 3.1.0
Wicket	Web application framework	1,672-2,578	2%-16%	106 k-165 k	0.05%-0.46%	1.3.0.beta1, 1.3.0.beta2, 1.5.3

- MCC. As illustrated in Figure 3, the ScottKnott ESD test reveals a hierarchical ranking structure consisting of four tiers. At the highest rank, we find GLANCE-LR, followed by GLANCE-EA and GLANCE-MD in the second and third positions, respectively. DeepLineDP occupies the fourth rank. This ranking alignment is supported by the MCC results presented in Table 2. Specifically, GLANCE-LR demonstrates the highest mean/median AUC values, while DeepLineDP records the lowest mean/median MCC values.

The key takeaway from the aforementioned observations is the unexpected finding that DeepLineDP does not exhibit superior file-level defect prediction performance in comparison to GLANCE. This outcome is particularly surprising in light of the following factors. GLANCE, a remarkably simplistic method, relies solely on the syntax features of source code, while DeepLineDP employs the cutting-edge deep learning technique known as HAN to model the hierarchical structure of source code, aiming to capture its semantic intricacies. Within our community, there has been a long-held belief centered on two fundamental premises: (1) semantic features possess a greater predictive capacity for defects

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

compared to syntax features; and (2) deep learning, as a powerful technique, excels in capturing the semantic nuances inherent in source code. Notably, in a prior study referenced as,[1] DeepLineDP emerged as the frontrunner, outperforming four state-of-the-art file-level defect prediction approaches: Bag of Words (BoW), Deep Belief Networks (DBN), Convolutional Neural Networks (CNN), and Bidirectional Long Short-Term Memory (BiLSTM). However, it is important to recognize that despite DeepLineDP's previous success in comparison to these techniques, the present findings indicate that it falls short of outperforming the comparatively simplistic GLANCE. This underscores the nuanced nature of software defect prediction, where the apparent advantage of semantic feature extraction through deep learning does not guarantee a consistent superiority over simpler methods based on syntax features. Further investigation and analysis are warranted to unravel the intricacies behind these unexpected results and to refine our understanding of the interplay between syntax and semantics in source code analysis for defect prediction.

5.2 | RQ2: cost-effectiveness in locating defective lines

Figures 4, 5, and 6 not only display the ScottKnott ESD rankings but also present visual representations of the corresponding distribution plots for three essential metrics: Recall@Top20%LOC, Effort@Top20%Recall, and IFA. To achieve a comprehensive and nuanced grasp of our findings, Table 3 provides a thorough and detailed breakdown of the performance metrics, shedding light on the intricate mechanisms underlying GLANCE and DeepLineDP. Within this section, we conduct a meticulous examination of these two approaches' performance across 14 target releases, encompassing a diverse array of nine open-source subject projects. In essence, Figures 4, 5, and 6 convey the visual narrative of our analysis, while Table 3 enriches our understanding by supplying the intricate details about GLANCE and DeepLineDP's predictive abilities in identifying defective lines.

From Figures 4, 5, 6, and Table 3, we have the following observations:

- Recall@Top20%LOC. As depicted in Figure 4, the results of the ScottKnott ESD test reveal a two-tier hierarchical ranking structure. GLANCE-EA, GLANCE-LR, and GLANCE-MD occupy the first tier, while DeepLineDP takes the second tier. Referencing Table 4, it is clear that regardless of whether the mean or median Recall@Top20%LOC is considered, GLANCE-EA, GLANCE-LR, and GLANCE-MD significantly outperform DeepLineDP. Strikingly, DeepLineDP's achieved recall is notably lower than the results documented in the original study.[1] A thorough

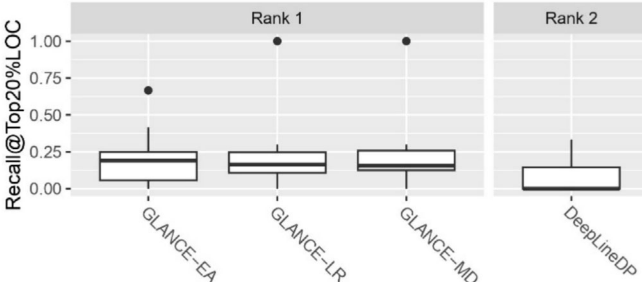


FIGURE 4 The ScottKnott ESD ranking on recall@Top20%LOC: DeepLineDP vs. GLANCE.

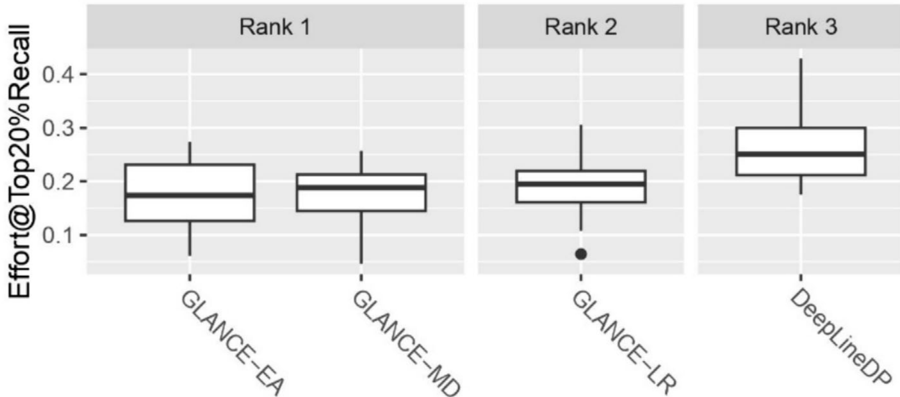


FIGURE 5 The ScottKnott ESD ranking on effort@Top20%recall: DeepLineDP vs. GLANCE.

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

FIGURE 6 The ScottKnott ESD ranking on IFA: DeepLineDP vs. GLANCE.

examination highlights that the inclusion of code comments in the performance evaluation, as expounded in Section 6.1, inflates the recall in the original study.

- Effort@Top20%Recall. Figure 5 illustrates a three-tier hierarchical ranking structure, determined by the ScottKnott ESD test. Notably, GLANCE-EA and GLANCE-MD are positioned at the top tier, followed by GLANCE-LR in the second tier, and DeepLineDP in the third. Assessing the Effort@Top20%Recall metric reveals that GLANCE exhibits superior performance compared to DeepLineDP. It is worth highlighting that both GLANCE-EA and

GLANCE-MD showcases an average Effort@Top20%Recall value of approximately 0.175, representing a substantial 34% reduction in comparison to DeepLineDP's score of 0.266. This finding is particularly intriguing, considering that GLANCE-EA and GLANCE-MD function as unsupervised classifiers, thus highlighting their independence from the training data.

- IFA. As depicted in Figure 6, the ScottKnott ESD test reveals a distinct ranking among the tested methods. Notably, GLANCE-MD secures the top position, with DeepLineDP following closely in second place and GLANCE-LR in third, while GLANCE-EA occupies the lowest rank. The findings from Table 3, which delve into the ranking of risky lines within each defective file, highlight that GLANCE-MD achieves a mean (median) IFA of 33.607 (25.500), while DeepLineDP records a mean (median) IFA of 41.464 (33.750). These results clearly demonstrate that the ranking of risky lines generated by GLANCE-MD surpasses that of DeepLineDP. Consequently, developers can anticipate reduced inspection effort when utilizing GLANCE-MD to identify initial defective lines.

The aforementioned observations emphasize a distinct and significant superiority of GLANCE over DeepLineDP in terms of line-level defect prediction performance. This notable performance disparity becomes particularly apparent when analyzing GLANCE's specialized variant, GLANCE-MD. Specifically, when comparing GLANCE-MD directly with DeepLineDP, GLANCE-MD emerges as the clear and compelling option for line-level defect prediction, demonstrating exceptional outcomes within a fixed 20% LOC inspection effort. Moreover, the implementation of GLANCE-MD significantly streamlines the process of identifying the initial defective line, effectively reducing the overall time required for defect localization. This revelation serves as compelling evidence that the hierarchical structure and comprehensive code context, adeptly captured by DeepLineDP, do not necessarily translate into a more cost-effective or efficient approach to defect localization when juxtaposed with the capabilities of GLANCE-MD. In essence, this highlights that while DeepLineDP excels in comprehending the intricate structural nuances of the code and the contextual relationships among tokens and lines, these advantages do not inherently result in a more streamlined or accurate process for pinpointing defective lines compared to the capabilities of GLANCE-MD. Consequently, the overarching conclusion here underscores that GLANCE-MD stands out as the superior choice for efficient and precise line-level defect prediction. This assertion underscores that in the domain of defect identification, simpler and more focused approaches can often outperform sophisticated and context-heavy methods such as DeepLineDP.

5.3 | RQ3: characteristics of detected defective lines

The numbers of randomly selected true defective lines identified by DeepLineDP, GLANCE-MD, GLANCE-EA, and GLANCE-LR are 371, 370, 365, and 366, respectively. Following in,[28] we classify true defective lines into the following five categories:

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 3 The accuracy for predicting defective files: GLANCE vs. DeepLineDP.

Project	Target release	AUC				Balanced accuracy				MCC			
		G-LR	G-MD	G-EA	DeepLineDP	G-LR	G-MD	G-EA	DeepLineDP	G-LR	G-MD	G-EA	DeepLineDP
ActiveMQ	5.2.0	0.756	**0.837**	**0.837**	0.745	0.687	0.713	**0.724**	0.598	**0.258**	0.129	0.205	0.084
	5.3.0	0.610	0.832	0.832	**0.833**	0.564	**0.717**	0.713	0.646	0.207	0.200	**0.291**	0.167
	5.8.0	0.730	0.849	0.849	**0.852**	0.672	0.727	**0.742**	0.669	**0.305**	0.179	0.287	0.158
Camel	2.10.0	0.636	**0.822**	**0.822**	0.764	0.687	0.681	**0.773**	0.665	**0.420**	0.095	0.189	0.135
	2.11.0	0.553	**0.797**	**0.797**	0.767	0.568	0.667	**0.735**	0.680	0.137	0.082	**0.153**	0.145
Derby	10.5.1.1	0.775	**0.826**	**0.826**	0.533	**0.795**	0.698	0.691	0.498	**0.404**	0.190	0.321	-0.015
Groovy	1.6.0.Beta_2	0.794	**0.825**	**0.825**	0.807	**0.771**	0.694	0.701	0.551	**0.520**	0.145	0.277	0.071
HBase	0.95.2	0.585	**0.705**	**0.705**	0.644	0.613	**0.643**	0.532	0.595	**0.215**	0.164	0.123	0.133
Hive	0.12.0	0.537	**0.752**	**0.752**	0.674	0.581	**0.682**	0.542	0.499	**0.204**	0.201	0.153	-0.004
JRuby	1.5	0.603	**0.908**	**0.908**	0.836	0.709	**0.755**	0.699	0.730	0.195	0.150	**0.265**	0.142
	1.7	0.622	**0.862**	**0.862**	0.809	0.528	**0.717**	0.645	0.701	0.057	0.136	**0.206**	0.129
Lucene	3.0.0	**0.784**	0.696	0.696	0.695	**0.793**	0.645	0.565	0.604	**0.482**	0.133	0.097	0.113
	3.1.0	0.506	**0.721**	**0.721**	0.683	0.519	**0.694**	0.581	0.553	0.025	**0.110**	0.072	0.037
Wicket	1.5.3	0.584	**0.841**	**0.841**	0.817	0.535	**0.748**	0.690	0.737	0.066	0.153	**0.179**	0.168
Mean		0.648	**0.805**	**0.805**	0.747	0.644	**0.699**	0.667	0.623	**0.250**	0.148	0.201	0.105
Median		0.616	**0.823**	**0.823**	0.766	0.643	**0.696**	0.695	0.625	**0.211**	0.148	0.197	0.131

*G-LR: GLANCE-LR, G-MD: GLANCE-MD, G-EA: GLANCE-EA.

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 4 The cost-effectiveness for locating defective lines: GLANCE vs. DeepLineDP.

		Recall@20%LOC				Effort@20% recall				IFA			
Project	Target release	G-LR	G-MD	G-EA	DeepLineDP	G-LR	G-MD	G-EA	DeepLineDP	G-LR	G-MD	G-EA	DeepLineDP
ActiveMQ	5.2.0	0.086	**0.133**	0.036	0.000	0.289	**0.203**	0.237	0.299	59	33.5	59	**24.5**
	5.3.0	**0.238**	0.000	0.000	0.000	**0.193**	0.257	0.274	0.323	**29**	42	62.5	30
	5.8.0	0.103	0.145	**0.164**	0.100	0.223	**0.203**	0.217	0.234	28	**21**	30	31
Camel	2.10.0	0.182	0.000	0.000	0.000	**0.197**	0.226	0.236	0.371	**23**	25	26.5	43.5
	2.11.0	0.200	**1.000**	0.667	0.000	0.156	**0.046**	0.062	0.272	38	**7**	9	23
Derby	10.5.1.1	0.000	0.000	0.000	0.000	**0.255**	0.256	0.260	0.264	77	73.5	191.5	**57.5**
Groovy	1.6.0.Beta_2	0.286	**0.300**	0.182	0.000	0.133	**0.107**	0.172	0.175	32	**12.5**	56	20
HBase	0.95.2	0.137	0.125	**0.253**	0.161	0.208	0.216	**0.141**	0.228	79.5	81	**76.5**	84
Hive	0.12.0	0.146	0.167	**0.219**	0.000	0.189	0.189	**0.122**	0.301	**24.5**	26	56	39
JRuby	1.5	0.000	0.125	**0.236**	0.000	0.306	0.184	**0.153**	0.237	114	**63**	116.5	70
	1.7	**0.250**	0.225	0.200	0.200	0.176	**0.144**	0.176	0.179	231	31	231	**29**
Lucene	3.0.0	0.300	0.270	**0.306**	0.184	0.064	0.110	**0.061**	0.207	**16**	**16**	17.5	20.5
	3.1.0	0.120	0.278	0.120	**0.333**	0.199	**0.188**	0.199	0.205	**21**	23.5	77	72
Wicket	1.5.3	**1.000**	0.215	0.417	0.000	**0.108**	0.150	0.119	0.429	38.5	**15.5**	18.5	36.5
Mean		**0.218**	0.213	0.200	0.070	0.193	0.177	**0.174**	0.266	57.893	**33.607**	73.393	41.464
Median		0.164	0.156	**0.191**	0.000	0.195	0.189	**0.174**	0.251	35.000	**25.500**	57.500	33.750

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

- *Computation*: computation-related defects involve changes to assignment statements, such as adding/deleting assignment statements or modifying equations, resulting in incorrect values being assigned to variables.
- *Data*: data-related defects typically involve changes to data declaration or initialization statements, such as modifications in variable types.
- *Interface*: interface-related defects involve issues with function definitions or dependencies on other functions, such as incorrect function definitions or faulty function calls.
- *Logic/control*: logic/control-related defects involve issues with the execution sequence or state of the software, further categorized into changes on loop statements, branch statements, and return/goto statements.
- *Others*: all other defects that cannot be classified into the above four categories, such as changes in preprocessor directives and code movement.

When classifying true defective lines into the above categories, the average Cohen's Kappa[21] coefficient is 0.81, In practical terms, a Kappa coefficient of 0.81 is typically considered indicative of "almost perfect" agreement.[29] Such a high level of agreement provides reliable assurance for the results of a study or task, allowing us to rely on these classification outcomes with confidence, indicating very good agreement in the classification between the two authors.

Table 5 reports the characteristics of true defective code lines identified by GLANCE and DeepLineDP, encompassing the defect size measured in tokens, alongside their mean absolute and relative positions within the files. Figure 7 illustrates the distributions of these true defective code lines across defect categories.

From Figure 7 and Table 5, we have the following observations:

1. Categories of defects. GLANCE and DeepLineDP consistently present a similar distribution of defect categories in their predictions, indicating shared patterns in identifying defect types. Notably, both models project defect category distributions with a greater prevalence of interface and logic/control defects, each accounting for over 30%. Interface defects hold a slight edge over logic/control defects, demonstrating a marginal advantage. The proportion of computation category defects is comparatively lower, approximately around 17%, while defects falling into the data and others categories exhibit even lower proportions, each below 15%.
2. Size of defects. In comparison to DeepLineDP, GLANCE more effectively identifies longer lines of code with increased token counts. This capability can be attributed to GLANCE's incorporation of NT as a contributing factor when computing defect-prone scores. Nonetheless, the marginal disparity in the number of tokens within true positive lines identified by both models suggests that although GLANCE takes into account the token count when evaluating lines of code for defects, its influence is relatively limited.
3. Location of defects. In terms of defect localization, both GLANCE and DeepLineDP consistently identify defects typically situated toward the middle to later sections of files. However, there is a notable contrast in the line numbers of code lines detected by GLANCE, which tend to be higher compared to those identified by DeepLineDP. This implies that GLANCE demonstrates greater proficiency in predicting defect-prone code lines within files containing a larger number of lines of code, as opposed to DeepLineDP. This variance may arise from GLANCE's consideration of the total number of source code lines within a file during the file classification phase. Files with a greater number of source code lines are regarded as more likely to harbor defects, thereby leading to a focus on defect prediction at the line level primarily within larger-scale files.

From the above observations, we can conclude that there is no substantial difference in the characteristics of detected true code lines when comparing GLANCE with DeepLineDP.

6 | DISCUSSION

In this section, we delve into the influence of code comments on the prediction performance of DeepLineDP, Furthermore, we analyze the contributions of features in GLANCE and examine the influence of file-level prediction on line-level defect prediction.

TABLE 5 The mean size and position of true defective code lines.

	Number of tokens	Line position number	Line position percentage
GLANCE-EA	4.64	7,804	53%
GLANCE-MD	4.78	6,230	56%
GLANCE-LR	4.95	7,317	54%
DeepLineDP	4.58	5,658	53%

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

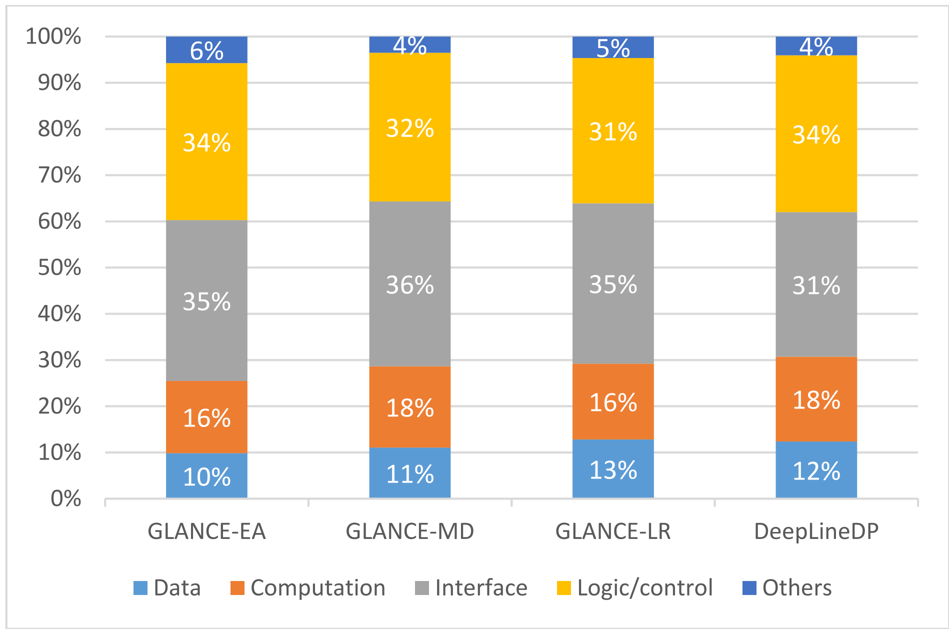


FIGURE 7 The distributions of identified true defective code lines over defect categories.

6.1 | Influence of code comments on DeepLineDP

For line-level defect prediction, it is customary to exclude comment lines. In,[1] during the evaluation of DeepLineDP, all comment lines were assigned a score of 0. Consequently, comment lines received the lowest rank when the code lines in a file were ranked. However, due to an unintended error in the original R script, comment lines were inadvertently included when counting the total number of code lines in a file. This led to unintentional inflation of the total number of code lines, resulting in an overestimation of defect prediction performance, including metrics like Recall@Top20%LOC and Effort@Top20%Recall. Table 6 provides an illustrative example of this scenario. As can be seen, when comment lines are excluded, the total number of code lines involved in ranking is 10. Consequently, Recall@Top20%LOC and Effort@Top20%Recall have values of 0 and 0.4, respectively. However, when comment lines are not excluded, the total number of code lines involved in ranking is 30. Consequently, Recall@Top20%LOC and Effort@Top20%Recall have values of 0.4 and 0.133, respectively. This inadvertent inclusion of comment lines significantly inflated the line-defect prediction performance.

The boxplots presented in Figure 8 offer a comprehensive overview of DeepLineDP's line-level defect prediction performance across the projects enlisted in Table 2, with and without the exclusion of comment lines. Strikingly, our observations closely parallel the findings documented in the initial research study,[1] highlighting the consistency and robustness of our results, especially concerning Recall@Top20%LOC and Effort@Top20%Recall when comment lines were taken into account. However, our investigation exposed a notable decline in the line-level defect prediction efficacy when comment lines were excluded, as depicted in Figure 8. Specifically, we observed a marked decrease in Recall@Top20% LOC, coupled with a substantial increase in Effort@Top20%Recall. This clear regression serves as a compelling testament to the pivotal role played by comment lines in reinforcing the predictive effectiveness of DeepLineDP. The discernible deterioration emphasizes the critical necessity of factoring in the exclusion of comment lines during evaluations, ensuring a meticulous and dependable assessment of the model's performance. This underscores the crucial significance of acknowledging the influence of comment lines, ultimately refining the precision and reliability of DeepLineDP's defect prediction capabilities.

6.2 | Contributions of features in GLANCE

RQ2 indicates that GLANCE outperforms DeepLineDP in line-level prediction, possibly due to its utilization and integration of three syntax features of code lines for defect identification. To understand why these three features enhance GLANCE's capability in line-level defect prediction, we conducted an ablation experiment to determine the effectiveness of individual features/heuristics. Table 7 summarizes the performance trends observed in GLANCE across various feature combinations. The first column outlines different feature combination settings employed in our ablation experiment. In the single-factor group (rows two to four), we exclusively focus on individual features. For example, "NT" indicates that GLANCE relies solely on the NT feature to identify defective code lines, while "NFC" implies exclusive consideration of the NFC feature, and

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

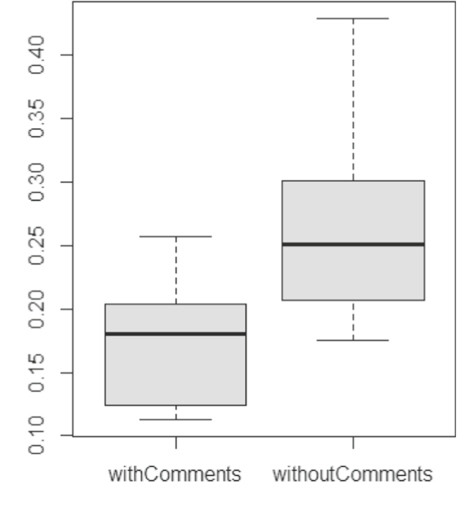
TABLE 6 Line-level defect prediction performance of DeepLineDP on "PreAddPartitionEvent.java" with/without the exclusion of comment lines.

	PreAddPartitionEvent.java	Without Comments		With Comments	
		Attention score	Rank	Attention score	Rank
1	/**			0	11
2	* Licensed to the Apache Software Foundation (ASF) under one			0	12
	...			0	...
17	*/			0	27
18					
19	package org.apache.hadoop.hive.metastore. events;	0.99994306	3	0.99994306	3
20					
21	import org.apache.hadoop.hive.metastore. HiveMetaStore.HMSHandler;	0.99996625	1	0.99996625	1
22	import org.apache.hadoop.hive.metastore.api. Partition;	0.99996621	2	0.99996621	2
23					
24	public class PreAddPartitionEvent extends PreEventContext {	0.99721108	5	0.99721108	5
25					
26	private final Partition partition;	0.99948066	4	0.99948066	4
27					
28	public PreAddPartitionEvent (Partition partition, HMSHandler handler) {	0.13964098	6	0.13964098	6
29	super (PreEventType.ADD_PARTITION, handler);	0.07722895	8	0.07722895	8
30	this. partition = partition;	0.08214881	7	0.08214881	7
31	}				
32					
33	/**			0	28
34	* @return the partition			0	29
35	*/			0	30
36	public Partition getPartition() {	0.06477406	9	0.06477406	9
37	return partition;	0.04723226	10	0.04723226	10
38	}				
39	}				
Recall@Top20%LOC		0		0.4 (better)	
Effort@Top20%Recall		0.4		0.133 (better)	
IFA		3		3	

"CE" suggests that only the CE feature is being exclusively considered. Moving to the two-factor group (rows five to seven), we individually exclude one feature from the three individual features in the single-factor group to examine their impact on GLANCE. For instance, "w/o NT" means that the NT feature is not used in defect identification, providing a perspective on GLANCE's line-level prediction effectiveness using only "NFC + CE". In the all-factor group (the last row), all three features are fused to formulate the GLANCE approach.

The findings presented in Table 7 offer significant insights into the effects of ablation con-figurations on GLANCE. First, in the single-factor group, solely using "CE" as the metric significantly diminishes the performance of line-level defect prediction. In contrast, exclusive utilization of "NT", while maintaining similar overall effectiveness, yields marginal improvements in GLANCE-specific metrics. Nevertheless, exclusive reliance on the "NFC" feature leads to enhancements across all three GLANCE features, indicating the sensitivity of NFC in identifying defective code lines. Second, in the two-factor group, compared to the single-factor group, diverse performance is observed in line-level defect prediction. Specifically, "w/o NFC", a significant decline is observed compared to with "NFC". Conversely, "w/o NT" and "w/o CE" exhibit significant and stable improvements over the single-factor groups of "NT" and "CE", respectively. Notably, "w/o CE" exhibits competitive performance surpassing even the best-performing "NFC", consistent with our findings from the single-factor analysis. Moreover, it underscores the complementary nature of NFC and NT in identifying defective code lines. Incorporating both NFC and NT into feature selection for further research on line-level defects, as facilitated by GLANCE, may offer promising avenues for future investigation. Third, integrating all three individual features in the single-factor group into GLANCE does not yield superior results for LL-SDP. Compared to "w/o NT", the inclusion of NT marginally reduces Recall@Top20%

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



(A) Recall@Top20%LOC (B) Effort@Top20%Recall

FIGURE 8 Comparison of the line-level defect prediction performance of DeepLineDP with and without the exclusion of comment lines.

TABLE 7 The median LL-SDP performance of GLANCE under different ablation settings (the performance values exceeding GLANCE are marked in **bold**).

	GLANCE-MD			GLANCE-EA			GLANCE-LR		
Feature fusion	R@20%	E@20%	IFA	R@20%	E@20%	IFA	R@20%	E@20%	IFA
NT	0.08	0.22	31.00	0.16	0.18	**42.00**	**0.20**	**0.16**	37.50
NFC	**0.19**	**0.17**	**25.00**	**0.24**	**0.15**	**40.00**	**0.27**	**0.15**	**31.50**
CE	0.00	0.29	42.50	0.05	0.25	90.25	0.13	0.29	58.00
w/o NT	**0.17**	0.19	**23.50**	**0.20**	**0.18**	57.50	**0.19**	0.20	43.25
w/o NFC	0.14	0.19	30.75	0.16	0.17	63.50	0.15	0.20	37.00
w/o CE	**0.29**	**0.15**	**19.50**	**0.30**	**0.12**	**36.25**	**0.30**	**0.12**	**30.75**
GLANCE	0.16	0.19	25.50	0.19	0.17	57.50	0.16	0.19	35.00

*R@20%: Recall@Top@20%LOC, E@20%: Effort@Top20%Recall.

LOC. In contrast, incorporating NFC results in improvements across all metrics compared to "w/o NFC". However, the addition of CE significantly lowers all performance metrics compared to "w/o CE". The potential exclusion of CE from GLANCE warrants careful consideration as a potential enhancement.

6.3 | Influence of file-level prediction on line-level defect prediction

The evaluation process in our study did not distinctly analyze file-level and line-level predictions. Rather, the line-level prediction may be influenced by file-level prediction outcomes. GLANCE and DeepLineDP both perform line-level defect prediction exclusively on the target defective files filtered out by their own file-level classifiers. Specifically, GLANCE's potential superiority over DeepLineDP in line-level prediction may stem from its robust file-level prediction performance. Notably, if DeepLineDP's file-level prediction matches GLANCE's performance, their line-level prediction performances might also align.

In the following, we investigate whether GLANCE's superior performance in line-level prediction compared to DeepLineDP is attributed to GLANCE's file-level prediction. We seek the answer to this question through the following two steps. In the file-level prediction stage, we utilize three versions of GLANCE's file-level classifier ManualDown, Effort Aware ManualDown, and Logistic Regression (short for MD, EA, and LR, respectively) for target defective files. Subsequently, in the line-level prediction stage, we apply DeepLineDP's line-level prediction to target defective files identified in the first stage. This approach results in three composite models based on the aforementioned strategy, named MD-DeepLineDP, EA-DeepLineDP, and LR-DeepLineDP.

Table 8 presents the line-level defect prediction performance for each version of GLANCE and its corresponding composite model. It is evident that every version of GLANCE surpasses its respective composite model. To clarify, when substituting DeepLineDP's line-level prediction for

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 8 The media performance of GLANCE and corresponding composite models (the best performance value of each indicator is marked in **bold** in each combination).

Approaches	Recall@TOP20%LOC(↑)	Effort@TOP20%recall(↓)	IFA(↓)
GLANCE-EA	**0.191**	**0.174**	**57.5**
EA-DeepLineDP	0.092	0.231	68
GLANCE-MD	**0.156**	**0.189**	**25.5**
MD-DeepLineDP	0	0.235	35.25
GLANCE-LR	**0.164**	**0.195**	**35**
LR-DeepLineDP	0	0.237	51.5

the original one in each GLANCE version, a noticeable decrease in line-level prediction performance occurs. For example, in the comparison between GLANCE-EA and EA-DeepLineDP, GLANCE-EA consistently outperforms EA-DeepLineDP across all three metrics. This observation underscores the interconnected nature of file-level and line-level predictions within defect prediction models. Consequently, we can deduce that GLANCE's superior performance in line-level prediction is not solely attributed to its excellence in file-level prediction.

7 | IMPLICATIONS FOR DEFECT PREDICTION COMMUNITY

The promise of deep learning for LL-SDP does not always translate into significant improvements. Our research challenges the widely-held assumption that applying deep learning will inherently lead to substantial advancements in LL-SDP. By demonstrating that GLANCE achieves comparable results to DeepLineDP, we provide a more nuanced perspective on the influence of deep learning in this field. This implies that the effectiveness of deep learning in LL-SDP could depend on multiple factors, and its advantages may not have universal applicability.

Depending exclusively on complex methods as the primary baselines could prove insufficient when evaluating LL-SDP. A comprehensive assessment requires the incorporation of a diverse set of techniques and methodologies beyond sophisticated approaches. Integrating straightforward yet efficient baselines in the evaluation of LL-SDP fosters a more thorough comprehension of its performance, taking into account not only the potential advantages but also the limitations and trade-offs inherent to deep learning within this field. Expanding our evaluation criteria enables us to make well-informed decisions about the most effective strategies for both developing and deploying LL-SDP solutions.

GLANCE should be considered as a highly practical and robust baseline for LL-SDP. By adopting GLANCE as a baseline, researchers and practitioners can enjoy several advantages. First, its simplicity in implementation ensures that it can serve as an accessible starting point for those new to the field, allowing them to quickly grasp the fundamentals of LL-SDP. Second, GLANCE's stellar performance serves as a dependable reference against which to gauge the effectiveness of more intricate approaches. In summary, GLANCE's simplicity, robustness, and strong performance make it an ideal choice as a baseline for LL-SDP.

There is a pressing imperative to delve deeper into the application of deep learning to enhance LL-SDP. We firmly hold the belief that deep learning harbors immense potential for bolstering the efficacy of LL-SDP, as evidenced by its remarkable successes in numerous other domains. Nevertheless, the existing research on deep learning-based LL-SDP remains insufficient, and there remains a substantial journey ahead to fully uncover the ways in which deep learning can be harnessed to enhance LL-SDP.

8 | THREATS TO VALIDITY

Construct validity is the degree to which the dependent and independent variables accurately measure the concept they purport to measure. In our study, we rely on the same dataset employed in the research conducted by Pornprasit et al.[1] Within this dataset, the dependent variable in use is a binary indicator that signifies whether a file or a code line is deemed defective. A paramount concern affecting construct validity is the potential existence of inaccurately labeled files or code lines. It is essential to note that these labels were acquired through the affected-version approach, a method that has been argued for its precision in recent studies when compared to alternative label collection techniques such as the time window approach and the SZZ-based approach.[18,19] However, it is imperative to acknowledge that, even with the advantages of the affected-version approach, the ongoing pursuit of more accurate datasets remains essential to mitigate this specific threat to construct validity in future research endeavors.

Internal validity is the degree to which conclusions can be drawn about the causal effect of independent variables on the dependent variables. In our study, a critical concern revolves around potential biases introduced during the implementation of DeepLineDP and GLANCE. To mitigate this threat to internal validity, we have taken deliberate steps. Specifically, we employed the source codes directly from their respective

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

replication packages. This approach is designed to enhance the rigor of our study by ensuring that the implementations of DeepLineDP and GLANCE align faithfully with their original designs, reducing the potential for implementation-related biases to affect our conclusions. By adopting this strategy, we bolster the internal validity of our research and enhance our ability to draw meaningful and reliable causal inferences from our findings.

External validity is the degree to which the results of the research can be generalized to the population under study and other research settings. In our experiment, we utilized a dataset comprising nine popular open-source Java projects with 32 distinct releases, encompassing a diverse array of application domains and project sizes. Our analysis of this dataset yielded consistent results in terms of performance comparisons across most projects. Consequently, we are reasonably confident that our findings remain within an acceptable margin of external validity. Nevertheless, it is essential to acknowledge that real-world project characteristics can be inherently stochastic and subject to change. As a result, we refrain from asserting that our results can be universally generalized to all projects. To address this potential limitation, future research should aim to replicate our study across a wider spectrum of projects, including those outside the Java ecosystem and in commercial settings. This broader scope of investigation will help mitigate potential external validity concerns.

9 | RELATED WORK

In this section, we provide an overview of the most pertinent previous research related to our work, encompassing fine-grained defect prediction and the exploration of the efficiency of deep learning in software engineering tasks.

9.1 | Enhancing defect prediction through fine-grained analysis

The field of defect prediction has witnessed a significant evolution over the past decades. Initially, it primarily centered around coarse-grained defect prediction, where defects were categorized at package, commit, file, and method levels.[7–9,30–36] Early methods heavily relied on manually crafted features, including product metrics and process metrics, to construct defect prediction models. However, these features often focused on specific module attributes and struggled to effectively capture their higher-level semantic relationships, consequently limiting their predictive capacity. To overcome this constraint, a multitude of deep learning techniques have been employed to autonomously generate semantic features from source code.[37–49] These approaches have laid a solid foundation for comprehending and mitigating software defects.

In recent years, research emphasis has shifted toward a more detailed and precise level: line-level defect prediction.[3–5,50–55] In,[3] a model interpretation technique[56] is used to predict buggy lines by interpreting risky code tokens within a defect prediction model. In,[4,5] natural language processing techniques are employed to determine whether a line contains defects. In the industry, various static analysis tools, including FindBugs,[52] PMD,[53] CheckStyle,[54] and Error Prone,[55] are utilized to identify potentially buggy lines in software systems. Fine-grained defect prediction significantly reduces the effort required for SQA. Developers can prioritize their testing and debugging efforts on the exact lines of code most likely to contain defects, thereby maximizing the efficiency of the entire software development lifecycle.

With numerous line-level defect identification approaches available, a natural question arises[1,2,57]: which approach is the most effective? In,[2] Guo et al assessed the current state of the field by analyzing the performance of state-of-the-art approaches and tools. The experimental results indicated that their proposed simple method, GLANCE, exhibited prediction performance comparable to or even superior to existing approaches and tools. Similarly, in,[1] Pornprasit et al reported that their proposed DeepLineDP outperformed n-gram as well as ErrorProne, establishing itself as the state-of-the-art approach. In our study, we rigorously compare GLANCE and DeepLineDP, and our findings demonstrate that GLANCE is a simpler yet effective approach.

9.2 | Examining the practical utility of deep learning models

With the rapid advancement of deep learning, a growing array of deep learning techniques are being introduced to address various challenges in software engineering. However, it is crucial to acknowledge that deep learning models typically come with a significant computational burden. This cost has raised concerns, particularly when these models are adopted without careful consideration of their practical value compared to traditional approaches. As a result, an emerging trend among researchers is to scrutinize whether existing deep learning models genuinely enhance the field of software engineering.

Remarkably, recent investigations have unveiled that many proposed deep learning models do not consistently outperform their traditional counterparts in software engineering tasks.[58–63] For instance, in,[58] Wu et al reported that SVM was 84 times faster than deep learning methods in determining relationships between questions in the Stack Overflow programmer discussion forum while maintaining accuracy, and sometimes achieving better results. In,[59] Liu et al found that a simple approach called NNGen was over 2,600 times faster than Neural Machine Translation

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

(NMT) but achieved higher accuracy. In,[60] Zeng et al discovered that a straightforward JIT defect prediction approach outperformed CC2Vec and DeepJIT and was 81 k X/120 k X faster in training and testing. In,[61] Lin et al reported that, for code comment updates, a heuristic-based approach called HEBCUP outperformed CUP in terms of accuracy while being over three orders of magnitude faster. These findings have significant implications for our software engineering community.

To ensure the judicious use of deep learning models, it is imperative that comprehensive comparisons are conducted between these models and conventional methodologies. This evaluation process serves as a litmus test to determine whether the adoption of deep learning genuinely provides benefits that outweigh the computational costs. Only when the advantages are clearly demonstrated should we integrate deep learning models into our software engineering practices.

10 | CONCLUSION

In this study, we conducted an in-depth exploration into the effectiveness and limitations of the cutting-edge deep line-level defect prediction approach, DeepLineDP. To achieve this, we utilized the same dataset and performance metrics as originally employed during the inception of DeepLineDP. This methodological consistency allowed us to conduct a rigorous comparative analysis between DeepLineDP and GLANCE, a recently introduced, simpler, traditional LL-SDP approach.

Our findings reveal that DeepLineDP does not outperform GLANCE, which is rather surprising. We initially anticipated that DeepLineDP, given the remarkable success of deep learning in various other fields, would significantly enhance LL-SDP when compared to GLANCE. However, this outcome suggests that the application of deep learning to LL-SDP does not invariably lead to substantial improvements. It underscores the fact that there is still much ground to cover in fully comprehending how deep learning can be effectively harnessed to augment LL-SDP.

In our upcoming studies, we have delineated two main avenues for investigation. Initially, we plan to replicate our ongoing experiment utilizing supplementary datasets to evaluate the extent of generalizability of our results. Secondly, we intend to delve deeper into the enhancement of DeepLineDP, aiming to fully unleash the potential of deep learning in LL-SDP. Potential strategies for this include the utilization of an extensive training dataset and the adjustment of the method to compute line scores, prioritizing suspicious lines for higher rankings.

Replication kit

The complete replication kit for this study is accessible through https://github.com/yuFeiCode/LL-SDP, allowing for convenient reproduction in future LL-SDP studies.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China under Grants 62162004, 62172205, and U21A20474, in part by the Guangxi Natural Science Foundation under Grant 2024GXNSFBA010248, 2018GXNSFDA138003, in part by the Basic Ability Enhancement Program for Young and Middle-aged Teachers of Guangxi under Grant 2023KY0057, and in part by Guangxi Collaborative Innovation Center of Multi-source Information Integration and Intelligent Processing.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in line-level-defect-prediction at https://github.com/awsm-research/line-level-defect-prediction.

ORCID

Xutong Liu https://orcid.org/0000-0002-3831-5505
Zhaoqiang Guo https://orcid.org/0000-0001-8971-5755
Yuming Zhou https://orcid.org/0000-0002-4645-2526
Junyan Qian https://orcid.org/0000-0002-1325-6975

ENDNOTE

* The number of samples randomly selected by the model is determined using the following method: First, calculate the total number of all random samples, then multiply this number by the ratio of the total number of TP (True Positives, i.e., correctly identified positive cases) lines in the current release to the total number of TP lines across all releases. If the calculated sample size for certain releases is not an integer, we round up the figure to ensure the feasibility and operability of the experiment.

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

REFERENCES

1. Pornprasit C, Tantithamthavorn C. DeepLineDP: towards a deep learning approach for line-level defect prediction. *IEEE Trans Softw Eng*. 2023;49(1):84-98. doi:10.1109/TSE.2022.3144348
2. Guo Z, Liu S, Liu X, et al. Code-line-level bugginess identification: how far have we come, and how far have we yet to go? *ACM Trans Softw Eng Methodol*. 2023;32(4):102. 1-55.
3. Wattanakriengkrai S, Thongtanunam P, Tantithamthavorn C, Hata H, Matsumoto K. Predicting defective lines using a model-agnostic technique. *IEEE Trans Softw Eng*. 2022;48(5):1480-1496.
4. Ray B, Hellendoorn V, Godhane S, Tu Z, Bacchelli A, Devanbu P. On the "naturalness" of buggy code. *ICSE*. 2016;428-439.
5. Wang S, Chollak D, Movshovitz-Attias D, Tan L. Bugram: bug detection with n-gram language models. *ASE*. 2016;708-719.
6. Zhou Y, Yang Y, Lu H, et al. How far we have progressed in the journey? An examination of cross-project defect prediction. *ACM Trans Softw Eng Methodol*. 2018;27(1):1-51.
7. Basili VR, Briand LC, Melo WL. A validation of object-oriented design metrics as quality indicators. *IEEE Trans Softw Eng*. 1996;22(10):751-761. doi:10.1109/32.544352
8. Hall T, Beecham S, Bowes D, Gray D, Counsell S. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans Softw Eng*. 2012;38(6):1276-1304. doi:10.1109/TSE.2011.103
9. Zhao Y, Damevski K, Chen H. A systematic survey of just-in-time software defect prediction. *ACM Comput Surv*. 2023;55(10):201:1-201:35.
10. Giger E, D'Ambros M, Pinzger M, Gall H. Method-level bug prediction. *ESEM*. 2012;171-180.
11. Pascarella L, Palomba F, Bacchelli A. Re-evaluating method-level bug prediction. *Saner*. 2018;592-601.
12. Pascarella L, Palomba F, Bacchelli A. On the performance of method-level bug prediction: a negative result. *J Syst Softw*. 2020;161:110493. doi:10.1016/j.jss.2019.110493
13. Trautsch A, Herbold S, Grabowski J. A longitudinal study of static analysis warning evolution and the effects of PMD on software quality in apache open source projects. *Empirical Software Engineering*. 2020;25(6):5137-5192. doi:10.1007/s10664-020-09880-1
14. Trautsch A, Herbold S, Grabowski J. Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction. *ICSME*. 2020;127-138.
15. Beller M, Bholanath R, McIntosh S, Zaidman A. Analyzing the state of static analysis: a large-scale evaluation in open source software. *Saner*. 2016;470-481.
16. Mikolov T, Sutskever I, Chen K, Corrado G, Dean J. Distributed representations of words and phrases and their compositionality. *Nips*. 2013;3111-3119.
17. Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate. 2014, arXiv:1409.0473.
18. Yatish S, Jiarpakdee J, Thongtanunam P, Tantithamthavorn C. Mining software defects: should we consider affected releases? *ICSE*. 2019;654-665.
19. Vandehei B, Costa D, Falessi D. Leveraging the defects life cycle to label affected versions and defective classes. *ACM Trans Softw Eng Methodol*. 2021;30(2):24:1-24:35.
20. https://www.calculator.net/sample-size-calculator.html
21. Cohen J. A coefficient of agreement for nominal scales. *Educ Psychol Meas*. 1960;20(1):37-46. doi:10.1177/001316446002000104
22. Hanley J, McNeil B. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology*. 1982;143(1):29-36. doi:10.1148/radiology.143.1.7063747
23. Yao J, Shepperd M. Assessing software defection prediction performance: why using the Matthews correlation coefficient matters. *Ease*. 2020;120-129.
24. Huang Q, Xia X, Lo D. Supervised vs unsupervised models: a holistic look at effort-aware just-in-time defect prediction. *ICSME*. 2017;159-170.
25. Parnin C, Orso A. Are automated debugging techniques actually helping programmers? *ISSTA*. 2011;199-209.
26. Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K. An empirical comparison of model validation techniques for defect prediction models. *IEEE Trans Softw Eng*. 2017;43(1):1-18. doi:10.1109/TSE.2016.2584050
27. Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K. The impact of automated parameter optimization on defect prediction models. *IEEE Trans Softw Eng*. 2019;45(7):683-711. doi:10.1109/TSE.2018.2794977
28. Zhao Y, Leung H, Yang Y, Zhou Y, Xu B. Towards an understanding of change types in bug fixing code[J]. *Inf Softw Technol*. 2017;86:37-53. doi:10.1016/j.infsof.2017.02.003
29. Landis JR, Koch GG. The measurement of observer agreement for categorical data. *Biometrics*. 1977;33(1):159-174. doi:10.2307/2529310
30. Zhou Y, Leung H. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Trans Softw Eng*. 2006;32(10):771-789. doi:10.1109/TSE.2006.102
31. Rahman F, Devanbu P. Ownership, experience and defects: a fine-grained study of authorship. *ICSE*. 2011;491-500.
32. Hassan AE. Predicting faults using the complexity of code changes. *ICSE*. 2009;78-88.
33. Nagappan N, Murphy B, Basili VR. The influence of organizational structure on software quality: an empirical case study. *ICSE*. 2008;521-530.
34. Liu Y, Li Y, Guo J, Zhou Y, Xu B. Connecting software metrics across versions to predict defects. *Saner*. 2018;232-243.
35. Yang Y, Zhou Y, Liu J, et al. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. *FSE*. 2016;157-168.
36. Zhou Y, Xu B, Leung H, Chen L. An in-depth study of the potentially confounding effect of class size in fault prediction. *ACM Trans Softw Eng Methodol*. 2014;23(1):1-51.
37. Xu Z, Li S, Xu J, et al. LDFR: learning deep feature representation for software defect prediction. *J Syst Softw*. 2019;158:110402. doi:10.1016/j.jss.2019.110402
38. Li J, He P, Zhu J, Lyu M. Software defect prediction via convolutional neural network. *QRS*. 2017;318-328.
39. Zhu K, Zhang N, Ying S, Zhu D. Within-project and cross-project just-in-time defect prediction based on denoising autoencoder and convolutional neural network. *IET Software*. 2020;14(3):185-195. doi:10.1049/iet-sen.2019.0278
40. Shi K, Lu Y, Chang J, Wei Z PathPair2Vec: an AST path pair-based code representation method for defect prediction. *J Comput Lang*. 2020;59:100979. doi:10.1016/j.cola.2020.100979

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

41. Wang S, Liu T, Nam J, Tan L. Deep semantic feature learning for software defect prediction. *IEEE Trans Softw Eng*. 2020;46(12):1267-1293. doi:10.1109/TSE.2018.2877612
42. Xu J, Ai J, Liu J, Shi T. ACGDP: an augmented code graph-based system for software defect prediction. *IEEE Trans Reliab*. 2022;71(2):850-864. doi:10.1109/TR.2022.3161581
43. Xu J, Wang F, Ai J. Defect prediction with semantics and context features of codes based on graph representation learning. *IEEE Trans Reliab*. 2021;70(2):613-625. doi:10.1109/TR.2020.3040191
44. Hoang T, Dam H, Kamei Y, Lo D, Ubayashi N. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. *MSR*. 2019;34-45.
45. Qiu S, Huang H, Jiang W, Zhang F, Zhou W. Defect prediction via tree-based encoding with hybrid granularity for software sustainability. *IEEE Trans Sustain Comput*. 2023, early access;1-12. doi:10.1109/TSUSC.2023.3248965
46. Xu Z, Zhao K, Zhang T, et al. Effort-aware just-in-time bug prediction for mobile apps via cross-triplet deep feature embedding. *IEEE Trans Reliab*. 2021;71(1):204-220. doi:10.1109/TR.2021.3066170
47. Chen J, Hu K, Yu Y, et al. Software visualization and deep transfer learning for effective software defect prediction. *ICSE*. 2020;578-589.
48. Wang H, Zhuang W, Zhang X. Software defect prediction based on gated hierarchical LSTMs. *IEEE Trans Reliab*. 2021;70(2):711-727. doi:10.1109/TR.2020.3047396
49. Wen M, Wu R, Cheung S. How well do change sequences predict defects? Sequence learning from software changes. *IEEE Trans Softw Eng*. 2020;46(11):1155-1175. doi:10.1109/TSE.2018.2876256
50. Majd A, Vahidi-Asl M, Khalilian A, Poorsarvi-Tehrani P, Haghighi H. SLDeep: statement-level software defect prediction using deep-learning model on static code features. *Exp Syst Appl*. 2020;147:113156. doi:10.1016/j.eswa.2019.113156
51. Rahman F, Khatri S, Barr E, Devanbu P. Comparing static bug finders and statistical prediction. *ICSE*. 2014;424-434.
52. FindBugs- Find Bugs in Java Programs. https://findbugs.sourceforge.net/
53. PMD. https://pmd.github.io/
54. Checkstyle. https://checkstyle.sourceforge.io/
55. Error Prone. https://errorprone.info/
56. Ribeiro M, Singh S, Guestrin C. "Why should I trust you?": explaining the predictions of any classifier. *KDD*. 2016;1135-1144.
57. Johnson B, Song Y, Murphy-Hill E, Bowdidge R. Why don't software developers use static analysis tools to find bugs? *ICSE*. 2013;672-681.
58. Fu W, Menzies T. Easy over hard: a case study on deep learning. *FSE*. 2017;49-60.
59. Liu Z, Xia X, Hassan A, Lo D, Xing Z, Wang X. Neural-machine-translation-based commit message generation: how far are we? *ASE*. 2018;373-384.
60. Zeng Z, Zhang Y, Zhang H, Zhang L. Deep just-in-time defect prediction: how far are we? *ISSTA*. 2021;427-438.
61. Lin B, Wang S, Liu K, Mao X, Bissyandé T. Automated comment update: how far are we? *ICPC*. 2021;36-46.
62. Zhang T, Han D, Vinayakarao V, et al. Duplicate bug report detection: how far are we? *ACM Trans Softw Eng Methodol*. 2023;32(4):97:1-97:32.
63. Majumder S, Balaji N, Brey K, Fu W, Menzies T. 500+ times faster than deep learning: a case study exploring faster methods for text mining stackoverflow. *MSR*. 2018;554-563.

How to cite this article: Zhou Y, Liu X, Guo Z, Zhou Y, Zhang C, Qian J. Deep learning or classical machine learning? An empirical study on line-level software defect prediction. *J Softw Evol Proc*. 2024;36(10):e2696. https://doi.org/10.1002/smr.2696

20477481, 2024, 10, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2696 by Tu Delft, Wiley Online Library on [08/04/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License