2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

Boosting Code-line-level Defect Prediction with Spectrum Information and Causality Analysis

Shiyu Sun, Yanhui Li*, Lin Chen, Yuming Zhou, Jianhua Zhao
State Key Laboratory for Novel Software Technology, Nanjing University, China
shiyusun@smail.nju.edu.cn, {yanhuili, lchen, zhouyuming, zhaojh}@nju.edu.cn

***Abstract*—Code-line-level defect prediction (CLDP) is an effective technique to incorporate comprehensive measures for buggy line identification to optimize efforts in Software Quality Assurance activities. Most CLDP methods either consider the textual information of the code or rely merely on file-level label information, which have not fully leveraged the essential information in the CLDP context, with historical *code-line-level labels* being incredibly overlooked in their application. Due to the vast number of code lines and the sparsity of the tokens they contain, leveraging historical code-line-level label information remains a significant challenge.**

To address this issue, we propose a novel CLDP method, Spectrum infOrmation and caUsality aNalysis based coDe-line-level defect prediction (SOUND). SOUND incorporates two key ideas: (a) it introduces a spectrum information perspective, utilizing labels from historical defective lines to quantify the contribution of tokens to line-level defects, and (b) it applies causal analysis to obtain a more systematic and comprehensive understanding of the causal relationships between tokens and defects. After conducting a comprehensive study involving 142 releases across 19 software projects, the experimental results demonstrate that our method significantly outperforms existing state-of-the-art (SOTA) CLDP baseline methods in terms of its ability to rank defective lines under three indicators, IFA, Recall@Top20%LOC, and Effort@Top20%Recall. Notably, in terms of IFA, our method achieves a score of 0 in most cases, indicating that the first line in the ranking list generated by our method is actually defective, significantly enhancing its practicality.

I. Introduction

Software Quality Assurance (SQA) is not just a practice but a cornerstone in software engineering, systematically ensuring the quality of software products [1]. Given the universal occurrence of defects in software systems [2], it is imperative to incorporate comprehensive measures for their prediction to optimize efforts in SQA activities [3]. Due to limited SQA resources [4], defect prediction approaches have been proposed at various granularities, with defect prediction models introduced at different levels, such as packages [5], components [2], modules [6], commits [7], and files [8]. Further research indicates that current defect prediction still needs to address the issue of insufficient granularity. For example, at the commonly used file-level defect prediction, researchers have found that within a defective file, only 1.2% to 2.5% of the code lines are defective [9], i.e., even if defect location can be accurately pinpointed at the file-level, programmers still need to spend a significant amount of human resources locating the defective code lines within the file.

Recently, researchers have started conducting defect prediction models at the code-line-level, a fine granularity, to produce more effective and actionable results. The main idea of existing code-line-level defect prediction (CLDP) approaches [9]–[12] is to convert the code line into a set of tokens and assume that lines containing higher suspicious tokens are potential defective lines. The methods to calculate the suspiciousness of tokens are divided into two folds. ① Unsupervised: these methods notice that defective code lines are either correlated with specific tokens, e.g., `for`, `if`, and `while` (i.e., tokens with control structures) applied in GLANCE [11], or blamed on unnatural token sequences, e.g., evaluating the average entropy value of tokens in N-gram [10]. ② Supervised: these methods try to learn the relations between defective files and tokens: they construct a file-level prediction model based on historical *file-level label* information to identify a file as defective or clean, then quantify the tokens' contribution to defect files, e.g., LineDP [9] and DeepLineDP [12]. We argue that all methods mentioned above have not fully leveraged the essential information in the CLDP context, with historical *code-line-level labels* being incredibly overlooked in their application. Due to the vast number of code lines and the sparsity of the tokens they contain [9], leveraging historical code-line-level label information remains a significant challenge.

To address this issue, we aim to construct a direct connection between tokens and code-line-level defects using historical code-line-level label information. The hidden logic of our connection is that, intuitively, *tokens prevalent in defective lines and rare in clean lines may serve as potential indicators of code-line-level defects*. Our method, **S**pectrum inf**O**rmation and ca**U**sality a**N**alysis based co**D**e-line-level defect prediction (SOUND), employs the following two ideas.

(a) **Spectrum information**: we employ suspicion score calculation formulas [13] based on spectrum information [14] to determine the connection of tokens to line-level defects. We design a novel angle to reshape tokens and line-level defects with the term of the program spectrum. We regard a code line as a code region consisting of tokens, and the label (i.e., defective/clean) of the line of code is regarded as the result (i.e., failed/passed) of running that code region. The appearance of a token in a defective/clean line implies that it has been *covered* once by a failed/passed execution.

(b) **Causality analysis**: to conduct a much deeper and more

* Yanhui Li is the corresponding author.

principled analysis of the causal relationship between tokens and defects, we employ causality analysis [15], Specifically, we consider the Bag of Tokens (BoT) [9], [11] and the defective/clean labels as variables to be analyzed in the causal graph [16]. Given the substantial computational resources required for causal analysis, we have restricted our study to the top tokens with the highest suspicion scores calculated from spectrum information.

Our method SOUND comprises the following four steps (see details in Section III): ① Preprocessing: we use BoTs as feature vectors representing source code files. ② Line-level Analysis: this step can be separated into spectrum information analysis and causality analysis, each providing the suspiciousness scores of tokens and the causal token set, respectively. ③ File-level Classification: we utilize file-level prediction models to exclude some files predicted as defect-free with the token features. ④ Global Ranking: we combine token suspicion score results and causal token results to enable effective prioritization of code lines.

To measure the performance of our method under CLDP scenarios, we select 19 project datasets for defect prediction and conduct cross-release predictions and validations based on 142 published versions of these projects [11]. To compare the quality of the ranking results, we choose three ranking performance metrics, IFA [17], Recall@Top20%LOC [18], and Effort@Top20%Recall [12] as the primary indicators for our experiments. We select five line-level defect prediction techniques as baselines, consisting of three SOTA methodologies (LineDP at TSE'2022 [9], GLANCE at TOSEM'2023 [11], and DeepLineDP at TSE'2023 [12]) and two typical approaches (N-gram [10] and ErrorProne [19]). The experimental results demonstrate that our method is actionable[1], which can effectively predict potential defective lines and significantly enhance ranking performance compared with the baselines.

The main contributions of this paper are listed as follows:

- **Method.** We introduce a novel defect prediction approach SOUND in the CLDP context that integrates the perspective of program spectrum analysis with causal analysis techniques to construct a direct connection between tokens and line-level defects.
- **Study.** Based on 19 defect prediction datasets, this research conducts a comprehensive empirical study encompassing 142 distinct releases. The experimental results demonstrate that our method is actionable and effective in identifying potential defect-prone lines and provides excellent ranking results.

II. Background and Motivation Example

In this section, we establish the groundwork by outlining the background of the CLDP approach, program spectrum, and causality analysis. Subsequently, we illustrate the main idea of our method with a motivating example.

[1]As stated in [17], developers are more likely to adopt the prediction model if its IFA is small. The results show that our method could achieve 0 under IFA on about half of the studied projects, which means our method is actionable.

A. Code-line-level Defect Prediction (CLDP)

Existing studies have posited that prioritizing software modules at a finer granularity optimizes cost-effectiveness [5], [20]. Consequently, CLDP would benefit SQA teams by allowing them to exert optimal effort in identifying and analyzing defects. The CLDP procedure deconstructs code lines to the token level for more detailed analysis and evaluates the suspiciousness of tokens, primarily characterized by three strategies [11]. ① **Static Analysis Tools.** ErrorProne [19], PMD [21], and CheckStyle [22] identify defective lines based on predefined rules. GLANCE [11] prioritizes the code lines by focusing on specific tokens: control elements, function calls, and the count of tokens per line. ② **Natural Language Processing-based Approach.** N-gram [10] ranks the code lines based on the mean entropy values of tokens within individual lines. ③ **Model Interpretation Technique-based (MIT) Approach.** LineDP [9] and DeepLineDP [12] separately utilize different techniques (LIME and attention mechanism) to quantify the tokens' contribution in the file to explain the defects.

B. Program Spectrum

Program spectrum, also known as code coverage, refers to the collection of executable statements covered and the execution result information gathered during the software testing process [23]. This kind of information can be employed for software fault localization [24]. Early studies [14], [25]–[27] focused solely on failing test cases in the area of spectrum-based fault localization. Later studies [28]–[30] revealed the above method's ineffectiveness, which led subsequent research to incorporate both successful and failing test cases, highlighting their differences.

Several effective formulas are commonly used for calculating suspicion scores in spectrum-based fault localization [31]. For example, empirical validation confirmed the effectiveness of the `Barinel` formula [13] calculating the suspiciousness of the statement s as

$$S_{\texttt{Barinel}}(s) = 1 - \mathit{passed}(s)/\{\mathit{passed}(s) + \mathit{failed}(s)\} \quad (1)$$

where $\mathit{passed}(s)/\mathit{failed}(s)$ means the number of passed/failed test cases that executed (i.e., covered) s.

C. Causality Analysis

The causality analysis provides an effective means of systematically and comprehensively understanding the causal relationships among variables [15]. Causal relationships denote the connections between two variables, where variations in one directly induce alterations in another [32], contrasting with the correlation that only reflects statistical associations between variables. Causal relationships are commonly represented using a causal graph [16]. This graph, often referred to as a Bayesian network or directed acyclic graph (DAG), comprises nodes and edges. Each node, for example X or Y, denotes a random variable, while an edge $X \rightarrow Y$ illustrates a causal mechanism indicating that X is a direct cause of Y.

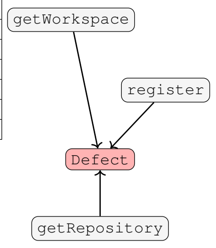
File_1
```
1  record = projectRealmCache.put( extensionRealms, projectRealm,
       extensionArtifactFilter );
2  projectRealmCache.register( project, record );
3  return record;
```
File_2
```
1  this.workspace = CacheUtils.getWorkspace(session);
2  this.localRepo = session.getLocalRepository();
3  this.repositories = new ArrayList<RemoteRepository>(repositories.
       size());
```
File_3
```
1  public static WorkspaceRepository getWorkspace(
       RepositorySystemSession session )
2  {
3     WorkspaceReader reader = session.getWorkspaceReader();
4     return (reader!=null) ? reader.getRepository() : null;
5  }
```
(a) Three historical files with defective lines (highlighted in red)

x	$defect(x)$	$clean(x)$	$S_{\texttt{Barinel}}(x)$
return	1	1	0.5
this	1	2	0.333
workspace	1	0	1
CacheUtils	1	0	1
getWorkspace	1	1	0.5
session	1	3	0.25
null	1	0	1
…	…	…	…

(b) Token suspiciousness

Files \ Tokens	return	this	…	Bug
File_1	1	0	…	1
File_2	0	3	…	1
File_3	1	0	…	1
…	…	…	…	…

(c) Graph learning data



(d) Causal graph

Fig. 1: An example of the construction of the CLDP model by our method SOUND, which contains (1) token suspiciousness scoring from spectrum information (see (b)) and (2) causal relationship identification from causal graphs (see (d)).

#line	Code	Score
1	`this.plugin = plugin.clone();`	0.333
2	`workspace = CacheUtils.getWorkspace( session );`	2.75
3	`this.localRepo = session.getLocalRepository();`	0.583
4	`CacheKey that = (CacheKey) o;`	0.0
5	`return CacheUtils.pluginEquals( plugin, that.plugin ) && eq( workspace, that.workspace ) && eq( localRepo, that.localRepo )`	3.5
6	`&& CacheUtils.repositoriesEquals( repositories, that.repositories ) && eq( filter, that.filter );`	1
7	`private static <T> boolean eq( T s1, T s2 ){`	0.0
8	`return s1 != null ? s1.equals( s2 ) : s2 == null;`	2.5
9	`}`	0.0

(a) A Java file with three defective lines (the ground truth) in red

Rank	#line	Score
1	5	3.5
2	2	2.75
3	8	2.5
4	6	1
5	3	0.583
6	1	0.333
7	4	0.0
8	7	0.0
9	9	0.0

(b) Initial result

Rank	#line	Score
1	2	2.75
2	5	3.5
3	8	2.5
4	6	1
5	3	0.583
6	1	0.333
7	4	0.0
8	7	0.0
9	9	0.0

(c) Enhanced result

Fig. 2: An example of applying CLDP models constructed by our method (in Figure 1) into a Java file.

Causal graphs are often unknown and need to be established through causal discovery from observational data. Causal discovery is inferring a DAG, which includes nodes for variables and edges for causal relationships. For instance, DiBS [33] is one SOTA score-based methodology using variational inference and is proven more efficient in the accurate learning of causal graphs and in excellent concordance with expert knowledge.

D. Motivation Example

We introduce a motivation example to demonstrate how our method works. Figures 1 and 2 illustrate how our model for token suspiciousness is constructed by our method and how it applies in actual cases to identify buggy lines in code files.

Figure 1(a) shows three code segments from three files in the historical release `mng-3.1.0`[2], with defective lines marked in red. As stated in Section I, we take a novel angle to consider tokens and line defects: ① a code line is divided as a sequence of tokens, and the label (defective/clean) of the line is regarded as the result (failed/passed) of running tokens; ② for token t, we count the appearance of t in defective/clean lines as the spectrum of t, denoted as $defect(t)/clean(t)$. We revise the `Barinel` formula (see Equation 1) by replacing $passed(s)/failed(s)$ with $defect(t)/clean(t)$ to calculate the suspiciousness of token t:

$$S_{\texttt{Barinel}}(t) = 1 - clean(t)/\{clean(t) + defect(t)\} \quad (2)$$

Figure 1(b) presents the results of token suspiciousness. To illustrate, the token `return` occurs twice: it appears once in Line 3 (clean) in the first code segment and once in Line 4 (defective) in the third code segment. As a result, $clean(\texttt{return}) = 1$ and $defect(\texttt{return}) = 1$, resulting in the suspicion score of `return`: $S_{\texttt{Barinel}}(\texttt{return}) = 1 - \frac{1}{2} = 0.5$. This result shows that the `Barinel` formula would help distinguish buggy tokens from clean ones, encouraging us to employ more spectrum-based formulas to calculate suspiciousness.

Moreover, as argued in Section I, we employ the causality analysis to further analyze the causal relationship between tokens and defects. Based on the calculated suspicion scores, we select the most suspicious tokens[3] to analyze their contribution to the defects. For these tokens, the frequency of their occurrences in each file and the label information of the files are extracted, as shown in Figure 1(c). Then the data is used by the causality analysis tool to construct a causal graph, as illustrated in Figure 1(d), which indicates that the causal

[2]The project `mng` is one of our studied 19 projects (see details in Table II).

[3]Due to computation limitation, we consider the top 100 tokens here.

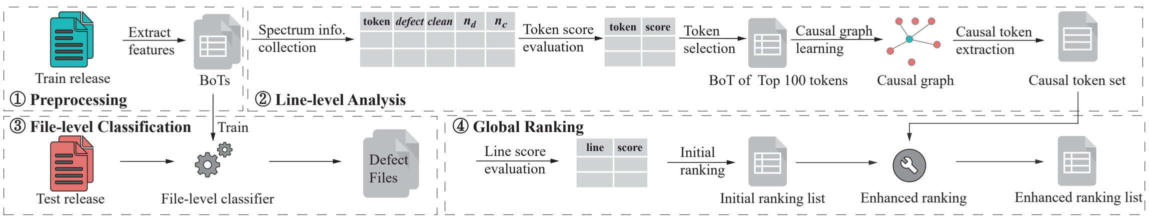


Fig. 3: The overview of our approach with four steps

tokens set is $\{$`getWorkspace`, `register`, `getRepository`$\}$, i.e., these tokens have the causal relationships with defects.

Figure 2(a) shows an actual defect prediction case within the file `DefaultPluginArtifactsCache.java` from the current release `mng-3.2.0`. We calculate the suspicion score of each line by summing the scores of the tokens it contains according to the token scores table calculated based on historical data in Figure 1(b), e.g., Line 2 contains four tokens, i.e., `workspace`, `CacheUtils`, `getWorkspace`, and `session`, resulting in its suspicion score as $1 + 1 + 0.5 + 0.25 = 2.75$.

We rank code lines with the following two strategies: ① The code lines are sorted in descending order based on their suspicion scores *without* considering causal tokens, yielding the *initial* results shown in Figure 2(b). ② This strategy employs the results of causal analysis to generate *enhanced* results. Lines that contain causal tokens are ranked first according to their suspicion scores, followed by lines without causal tokens. In this case, Line 2 contains the token `getWorkspace`, which is causally related to the defect (see Figure 1(d)), thereby increasing its priority and placed at the top, as shown in Figure 2(c). Comparing the result from ① and ②, we observe that Line 2 (defective) is ranked first by adding causal relationships (the IFA[4] value equals to 0 here), i.e., causal relationship adjustments refine the ranking result. We conclude that causality analysis would strengthen the prediction results, especially at the top of the ranking list.

III. Methodology

This section will detail our approach, as shown in Figure 3, which consists of four parts: preprocessing, line-level analysis, file-level classification, and global ranking.

A. Preprocessing

In this step, we employ Bag of Tokens (BoT) as features representing the source code files. Specifically, we utilize the `CountVectorizer` [34] function from the `Scikit-Learn` library to extract code tokens from each code line in the historical source code, i.e., convert Line l into its token sequence $token(l)$. Considering that the programming language in the source code is case-sensitive (e.g., two variables, `var` and `Var`, are noticeably different), no lowercase conversion, stemming, or tokenization processing is implemented. This decision ensures that all tokens in the files, excluding non-alphabetic characters, are retained during preprocessing to preserve meaningful tokens. To illustrate, Line 2 in Figure 2(a) is divided into four tokens, i.e., `workspace`, `CacheUtils`, `getWorkspace`, and `session`, after preprocessing.

[4]IFA [17] is one of our three performance indicators (see details in Section IV-D). Less IFA values mean that our method is more actionable.

TABLE I: Details of five applied formulas

Formulas	
`Barinel` [35]:	$S_{\texttt{Barinel}}(t) = 1 - \frac{clean(t)}{clean(t)+defect(t)}$
`Dstar` [36]:	$S_{\texttt{Dstar}}(t) = \frac{defect(t)^*}{clean(t)+n_d-defect(t)}$
`Ochiai` [37]:	$S_{\texttt{Ochiai}}(t) = \frac{defect(t)}{\sqrt{n_d(defect(t)+clean(t))}}$
`Op2` [38]:	$S_{\texttt{Op2}}(t) = defect(t) - \frac{clean(t)}{n_c+1}$
`Tarantula` [39]:	$S_{\texttt{Tarantula}}(t) = \frac{defect(t)/n_d}{defect(t)/n_d+clean(t)/n_c}$

[a] According to prior research [13], we use $* = 2$.

B. Line-level Analysis

This section presents the methodology for conducting line-level analysis of the source code, divided into two parts: spectrum information calculation and causality analysis.

1) Spectrum Information Calculation: We introduce the program spectrum perspective to quantify the contribution of tokens to the defect lines. We view individual lines of code as the set of program statements covered during test executions, the tokens they contain as program statements, and the line-level label as the results of test case executions.

To facilitate the following calculation, we list four notations (some have already been applied in Equation 2) as follows:

- $defect(t)$: number of defective lines containing token t
- $clean(t)$: number of clean lines containing token t
- n_d: total number of defective lines
- n_c: total number of clean lines

Based on these notations, the token suspiciousness calculation process comprises these steps:

① **Spectrum Information Collection**. We collect spectrum information for each token by iterating through each line of historical source code and updating the spectrum information for the tokens contained in each line based on their actual defect status. Then we get the spectrum information (i.e., $defect(t)$ and $clean(t)$) for each token t, along with the counts (i.e., n_d and n_c) of defective lines and clean lines.

② **Token Score Evaluation.** With the collected spectrum information, we calculate the suspicion score for each token. Based on previous research [31], we predominantly encompass several commonly utilized formulas for calculating suspicion scores in spectrum-based fault localization. Specifically, we employ five formulas, i.e., `Barinel`, `Dstar`, `Ochiai`, `Op2`, and `Tarantula`, as shown in Table I, to calculate the suspicion score for each token. For each formula, we revise its variables in the context of CLDP, i.e., replacing original variables with four notations defined above (similar to the conversion from Equation 1 to Equation 2).

2) Causality Analysis: Causality analysis effectively breaks down complex relationships among different factors, presenting them as an intuitive and highly interpretable causal graph [40]. The edges within causal graphs signify causal relations differentiated from widely recognized correlations. The fundamental contrast between causal relations and correlations necessitates the use of causality analysis. The process can be divided into three steps:

① **Token Selection.** Given the substantial computational resources required for causal analysis, we have restricted our study to the top 100 tokens[5] with the highest suspicion scores.

② **Causal Graph Learning.** We extract the BoTs of the 100 tokens with the corresponding label of each file to learn the causal graph. By utilizing the `ScaleFreeDAGDistribution` and `BGe` in `dibs.models` [41] as graph model and observation model, we have the result of the adjacency matrix to denote the DAG, where each element denotes the causal probability of an edge between two variables.

③ **Causal Token Extraction.** We retain the tokens with causal probabilities ≥ 0.99 as the set of tokens causally related to defect situations since the probabilities obtained by the model are not necessarily 0 or 1 [41].

C. File-level Classification

To enhance prediction accuracy, our method utilizes a file-level prediction tool to exclude some files predicted as defect-free. Research [9] has shown that logistic regression (LR) is a simple yet effective supervised classifier capable of accurately identifying defective lines in defect prediction scenarios.

In this approach, the LR classifier employs BoT vectors of files generated from preprocessing as features [11], with the defect status of files as labels trained on historical data. After training, the classifier is applied to current files for classification, yielding predicted labels and file scores for subsequent ranking, as detailed in Section III-D.

D. Global Ranking

Previous studies [9], [11], [12] have either investigated the effectiveness of line ranking within a single file or first ranked the files based on their defect probability, followed by ranking the lines within single files and then assessing the effectiveness of the overall ranking[6]. Our approach utilizes a more effective strategy, i.e., a global ranking strategy that integrates lines from all current files and ranks them based on line-level suspicion scores[7]. As explained in Section III-C, we first use LR to filter the code files so that only the predicted defective files remain. For these files, excluding comment lines, we then perform a subsequent ranking of source code lines.

① **Line score evaluation.** Based on the suspicion score obtained for each token, for lines l in the current files, the scores of each token contained within Line l are accumulated according to the occurrence times of the token in that line:

$$S_f(l) = \sum_{t \in token(l)} S_f(t) \times freq(t) \tag{3}$$

where $f \in \{\texttt{Barinel}, \texttt{Dstar}, \texttt{Ochiai}, \texttt{Op2}, \texttt{Tarantula}\}$ and $freq(t)$ indicates the occurrence times of the token t in Line l. This process yields the line-level suspicion score $S_f(l)$ for each code line in the current files.

② **Initial ranking.** The source code lines are ranked in reverse order based on the line-level suspicion score results, discarding those with a suspicion score of 0, to yield the initial ranking list.

③ **Enhanced ranking.** We utilize a set of causal tokens extracted from historical data (see Section III-B) to enhance the ranking performance at the top of the list. The defect rate $x\% = \frac{n_d}{n_d + n_c}$ is computed from the historical data. For the top $x\%$ part of the current ranking list, lines that contain causal tokens will be moved to the front of the ranking list. For instance, the top $x\%$ part of the ranking list based on initial line suspicion scores is $[l_1, l_2, l_3, l_4, l_5]$, where l_1 has the highest score and l_5 has the lowest. Assuming l_2 and l_4 contain causal tokens, the result after tuning is $[l_2, l_4, l_1, l_3, l_5]$.

IV. Experimental Setups

This section details the experimental setups, including datasets, settings, baselines, and evaluation indicators.

A. Dataset

This paper uses an extensive line-level defect dataset collected by Guo et al. [11], including 142 releases from 19 projects. The dataset provides both file-level and line-level defect data. File-level data includes filenames, defect labels, and source code, while line-level data contains lists of defective files and the corresponding line numbers. Table II shows the details of 19 studied projects. These projects display considerable variation in file-level and line-level metrics, allowing for a thorough evaluation of our methods' effectiveness.

B. Experimental settings

Prediction setting. We employ the cross-release (CR) prediction setting [11], [42] to perform our experiments. (a) Our method arranges the n releases of each project in chronological order. (b) Within the project, using the file-level and line-level defect labels provided by the dataset from release i $(i \in [1, n-1])$, we train models to predict defects for the

[5] We consider the top 100 tokens the primary representative information across releases. If we extend the token number, we argue that the potential overfitting would increase. Due to space limitation, we present the comparison results between top 100 and top 200 tokens in the online appendix (see Section IX-A).

[6] For example, given two files A and B with defect probabilities 0.8 and 0.9, each has two lines, denoted as l_1^A/l_2^A (with line-level suspicion 1/3) and l_1^B/l_2^B (with line-level suspicion 2/4). The existing method would output the ranking list: $\{l_2^B, l_1^B, l_2^A, l_1^A\}$, i.e., it first ranks files (the defect probability of file B is more significant than that of A), and then rank lines within the files according to their suspicion values.

[7] For the examples illustrated in Footnote 6, our approach outputs: $\{l_2^B$ (with the suspicion value 4), l_2^A (with 3), l_1^B (with 2), l_1^A (with 1)$\}$, i.e., only focuses on the suspicion values of lines. We will compare our strategy with the original ones in Section VI-B.

TABLE II: Details of 19 Studied Projects

Project	#Release	Avg. #Files	Avg. #Lines[a]	Avg. %Buggy Lines
ambari	7	2307	366.3	1.47%
amq	14	1835	292.8	1.13%
bookkeeper	3	240	57.7	3.81%
calcite	8	1430	304.5	4.20%
cassandra	6	583	155.7	1.22%
flink	2	3583	564.1	0.20%
groovy	14	914	194.1	0.62%
hbase	5	1076	487	0.94%
hive	4	3263	953.1	0.29%
ignite	3	2658	612	1.68%
log4j2	11	798	109	1.56%
mahout	6	933	134.7	1.63%
mng	6	663	94.6	0.63%
nifi	4	2809	432.1	1.37%
nutch	13	332	56.7	1.06%
storm	5	997	166.4	0.24%
tika	12	393	66.2	2.11%
ww	15	1087	150.8	0.86%
zookeeper	4	339	63.7	1.26%

[a] The unit of the average number of rows: KLOC

next release $i+1$. This results in 123 (i.e., 142-19) release pairs $\langle i, i+1\rangle$. We analyze individual results and then compute the mean value, thereby benchmarking the performance of our method on each project. We employ the historical data from the **SPR** (Single Prior Release) for training because it is widely used in previous studies [9], [43]–[46]. Amasaki et al. [47] found SPR with the best prediction performance for CR defect prediction. We analyze that there may be concept drift [48], [49] between too-old previous releases and the predicted release, i.e., the distribution of buggy lines in the old releases deviates from the predicted release.

Ranking setting. We combine code lines from every risky file for global ranking. (a) Code lines are ranked based on the line-level suspicion scores. (b) The file-level score from LR would be considered to rank lines with the same score. For example, file A with a file-level score of 0.8, contains lines l_1^A/l_2^A with line-level scores 9/8, and file B with a file-level score of 0.9, contains lines l_1^B/l_2^B with line-level scores 8/6. The ranking list is $[l_1^A, l_1^B, l_2^A, l_2^B]$, with the line-level scores $[9, 8, 8, 6]$, where l_1^B is listed before l_2^A for file B is more likely to be defective.

C. Baselines

We select five CLDP techniques as baselines, consisting of three SOTA methodologies and two typical approaches.

(a) GLANCE (TOSEM'2023) [11] is an approach that incorporates control elements and statement complexity. There are three variants based on the type of file-level classifier: GLANCE-LR, GLANCE-MD, and GLANCE-EA.

(b) DeepLineDP (TSE'2023) [50] is a complex deep learning approach that automatically extracts the semantic features of the surrounding tokens and lines to achieve the prediction of file- and line-level defects.

(c) LineDP (TSE'2022) [9] is a novel approach that utilizes a model-agnostic technique LIME, to explain the file-level classifier, extracting the most suspicious tokens.

(d) N-gram [10], [51], [52] is a typical natural language processing-based approach, using the entropy value to infer the naturalness of each code token. Defective lines are assessed based on the average entropy values of tokens within individual lines.

(e) ErrorProne [19] is a static analysis tool from the Google company that extends a primary Java compiler (javac) to identify defects in code lines using error-prone guidelines.

The ranking strategy of baselines is modified based on our ranking setting in Section IV-B. Since N-gram and ErrorProne do not provide priority for the files, we ranked the lines based on the file-level prediction probabilities provided by DeepLineDP, which reported better than common file-level classifiers [50]. Moreover, to fulfill the prediction setting in Section IV-B, we have configured the DeepLineDP method such that 50% of the data from the preceding release is allocated to the training set. In contrast, the remaining 50% is the validation set and executed across five experimental iterations to reduce the impact of randomness.

D. Evaluation indicators

Since our method is proposed to help improve SQA performance under limited resources, we focus on the effectiveness of ranking, especially at the *top* of the list. We select three metrics extensively employed in prior studies [9], [11], [12].

(a) **IFA** [17] (Initial False Alarm) measures the number of clean lines tested before identifying the first defective line. Lower IFA implies less effort expended on testing non-defective lines before the first detective line.

(b) **Recall@Top20%LOC** [18] (Recall@20% for short) assesses the percentage of total defective lines found within the first 20% of code lines in a given release. High Recall@20% values indicate that the method successfully prioritizes most defective lines within the top 20%, enabling the discovery of a large number of defects with limited testing effort.

(b) **Effort@Top20%Recall** [12] (Effort@20% for short) measures the proportion of code lines tested to discover 20% of the total defective lines. Lower Effort@20% values indicate that developers can identify the top 20% actual defective lines with minimal cost.

V. Results

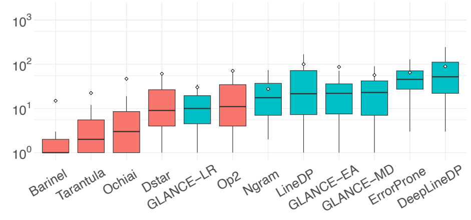
This section presents the results of the experiments with four research questions (RQs).

A. RQ1: How Effective Does Our Method Perform Compared against Baselines in Ranking Defective Lines?

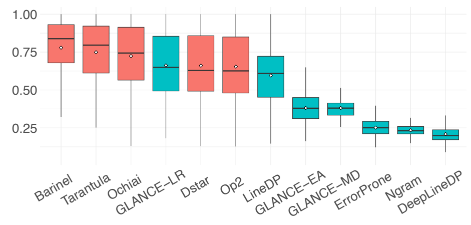
Motivation&Approach. This RQ aims to validate the effectiveness of our method in CLDP. Our method is benchmarked against the baseline methods (see Section IV-C) recognized for their effectiveness in defect prediction studies.

To address RQ1, we use the CR prediction setting (see Section IV-B). All studied (our and baseline) methods are applied to rank code lines in 123 releases from 19 projects, producing 123 line-level prediction results. By analyzing the IFA, Recall@20%, and Effort@20% metrics (see Section IV-D), we assess the performance improvements of our proposed methods compared to the baselines.

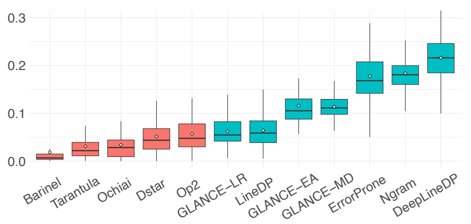
Results. Figure 4 illustrates the overall results of our method and baselines with the boxplots under three indicators sorted by their *median* values. ① Figure 4(a) indicates that our methods, using five different formulas, hold four out of the



(a) IFA (↓) boxplots in the increasing order



(b) Recall@20% (↑) boxplots in the decreasing order



(c) Effort@20% (↓) boxplots in the increasing order

Fig. 4: The Boxplots of our methods marked red and baseline methods marked green, where ↑/↓ indicates that a higher/lower value of the metric is better.

top five positions under the median of IFA. Remarkably, the `Barinel`-based method secures the top rank with an IFA score of 0. With the formula `Barinel`, it is observed that the first line of the ranking list is defective in most cases. ② Figure 4(b) presents that among the top five positions in terms of the median of Recall@20%, three are occupied by our methods. Except for the baseline GLANCE-LR, our methods demonstrate superiority over the other six baselines. ③ Figure 4(c) shows that, at the median, our methods achieve Effort@20% values of 0.007-0.048, occupying the top five, while baselines spend more effort with medians of 0.055-0.216 to discover 20% defective lines.

Table III presents the *average* values of IFA, Recall@20%, and Effort@20% for the methods on **19 projects**. We examine the best-performing methods marked gray for each project (see the “#Best/19” column) to highlight the superiority of our approach. ① Our methods demonstrate a superior IFA performance in 13/19 projects, match the best baseline in 2 projects, and are only outperformed by the baselines in 4 projects. The `Barinel`-based method has performed the best, achieving the best IFA results in 11/19 projects, with 10 of them achieving a value of 0, and is also the best with a total average value of 13. ② Regarding Recall@20%, our proposed methods achieve the best performance in 17/19 projects and are surpassed by the baselines in just two projects. The `Barinel`-based method has still demonstrated superior performance, securing the highest Recall@20% results in 16/19 projects, and the average value of 0.779 for the entire dataset is also the best. ③ Regarding Effort@20%, our approaches excel in 16/19 projects and are exceeded in merely three projects. The `Barinel`-based method shows the best performance, obtaining the best Effort@20% results in 16/19 projects, and overall, has the best performance, with the average value of 0.019.

Answer to **RQ1**. Our methods with five formulas demonstrate a considerable improvement in the performance of ranking defective lines over the baselines. The `Barinel`-based method has performed the best.

B. *RQ2: How Does the File-level Classifier Affect the Performance of Our Methods?*

Motivation&Approach. The proposed methods in this study combine file-level defect predictions. They employ an LR classifier at the file level to eliminate many non-defective lines from non-defective files, reducing the effort required to construct line-level prediction models (see Section III-C). This RQ aims to compare the performance with and without using the file-level classification to check its impact on our method.

Results. The detailed mean and median values are presented in Table IV. In terms of IFA, under 9/10 scenarios (10 scenarios = 5 formulas $\times$ mean/median values), our methods with the file-level classification perform better, with the best case showing 66% ($\frac{200-69}{200}$) and the worst case showing 13% ($\frac{15-13}{15}$) enhancement, and match under the remaining one scenario with the median IFA value 0, which already has the best performance. Moreover, our methods (with) demonstrate superior Recall@20% performance under all five formulas in all ten scenarios, with the best mean value of 0.779 and median value of 0.838 under `Barinel`, our best-performed formula. The greatest enhancement is 0.288 at the mean and 0.276 at the median under `Op2`. Regarding Effort@20%, our approach wins in 8/10 scenarios, matches in one, and loses to the modified one (i.e., without) in one scenario by a marginal difference of 0.001. Under the `Op2` formula, the most significant improvement in the average value is 0.045. The minimal improvement is 0.011 at the mean under `Tarantula`.

Answer to **RQ2**. Utilizing a file-level classifier enhances the ranking performance at the top, especially under the IFA and Recall@20% indicators.

C. *RQ3: How Do the Causality Analysis Enhance the Ranking Performance?*

Motivation&Approach. In our methodology, we use causality analysis to conduct a much deeper and more principled analysis of the causal relationship between tokens and defects. In this section, we will analyze the effect of causality analysis.

TABLE III: The average values of our methods with five formulas and baseline methods on 19 studied projects under three indicators. Due to space limitations, we employ the first three letters of the project name to represent it, e.g., amb is short for the project ambari. The gray background indicates that this method is the best.

		amb	amq	boo	cal	cas	fli	gro	hba	hiv	ign	log	mah	mng	nif	nut	sto	tik	ww	zoo	#Best/19
IFA (↓)	GLANCE-LR	11	7	94	10	33	278	48	18	341	0	10	12	15	6	14	29	1	22	37	2
	GLANCE-EA	26	19	114	19	35	202	148	787	633	0	30	18	29	6	20	174	0	49	74	2
	GLANCE-MD	13	23	86	11	49	96	170	29	152	1	30	28	29	9	31	264	0	51	103	1
	LineDP	29	17	155	11	342	336	137	245	649	8	118	13	17	61	26	97	164	11	8	0
	DeepLineDP	34	34	312	24	42	44	98	131	79	13	77	55	161	16	179	43	151	58	262	0
	N-gram	40	22	4	4	3	43	19	27	26	7	28	18	61	18	32	12	15	29	200	3
	ErrorProne	32	53	51	15	92	32	115	196	92	65	37	54	132	109	48	50	48	39	138	0
	Barinel	53	0	116	0	0	371	33	0	76	0	0	1	0	0	0	9	0	1	12	11
	Dstar	237	8	129	7	147	103	26	21	882	32	54	30	3	10	6	21	7	6	153	0
	Ochiai	80	2	342	11	28	61	71	102	603	3	6	2	0	1	1	6	30	14	120	2
	Op2	268	11	256	8	154	167	38	160	544	12	43	65	24	46	7	118	32	8	150	0
	Tarantula	43	3	167	1	9	21	31	206	53	1	2	0	3	6	0	25	27	1	7	5
Recall@20% (↑)	GLANCE-LR	0.553	0.834	0.336	0.425	0.608	0.335	0.685	0.513	0.436	0.465	0.644	0.678	0.936	0.562	0.699	0.690	0.603	0.820	0.542	1
	GLANCE-EA	0.425	0.516	0.230	0.289	0.220	0.477	0.413	0.205	0.301	0.423	0.368	0.357	0.544	0.364	0.331	0.291	0.374	0.419	0.415	1
	GLANCE-MD	0.416	0.416	0.315	0.319	0.301	0.423	0.392	0.338	0.337	0.359	0.361	0.373	0.419	0.366	0.411	0.362	0.362	0.396	0.465	0
	LineDP	0.475	0.677	0.286	0.424	0.560	0.258	0.539	0.444	0.453	0.460	0.605	0.557	0.802	0.538	0.673	0.712	0.561	0.769	0.474	0
	DeepLineDP	0.253	0.197	0.229	0.209	0.221	0.244	0.301	0.179	0.262	0.176	0.184	0.172	0.202	0.265	0.169	0.182	0.155	0.214	0.202	0
	N-gram	0.198	0.223	0.237	0.247	0.286	0.370	0.240	0.247	0.197	0.284	0.274	0.212	0.252	0.248	0.197	0.255	0.227	0.241	0.163	0
	ErrorProne	0.273	0.235	0.232	0.278	0.260	0.175	0.306	0.211	0.285	0.196	0.230	0.184	0.171	0.253	0.178	0.272	0.252	0.319	0.203	0
	Barinel	0.614	0.860	0.339	0.753	0.819	0.338	0.677	0.751	0.475	0.669	0.836	0.765	0.961	0.738	0.901	0.788	0.801	0.848	0.719	16
	Dstar	0.516	0.763	0.286	0.395	0.655	0.345	0.631	0.532	0.410	0.428	0.677	0.699	0.953	0.613	0.784	0.682	0.607	0.823	0.538	0
	Ochiai	0.555	0.821	0.294	0.500	0.758	0.279	0.664	0.655	0.415	0.541	0.755	0.751	0.961	0.685	0.870	0.731	0.729	0.838	0.663	1
	Op2	0.519	0.764	0.263	0.384	0.644	0.274	0.631	0.497	0.391	0.415	0.652	0.703	0.953	0.610	0.769	0.666	0.62	0.821	0.550	0
	Tarantula	0.595	0.846	0.342	0.599	0.795	0.279	0.671	0.724	0.432	0.603	0.790	0.744	0.961	0.719	0.876	0.761	0.750	0.845	0.688	2
Effort@20% (↓)	GLANCE-LR	0.087	0.039	0.125	0.102	0.087	0.124	0.057	0.095	0.056	0.089	0.058	0.049	0.051	0.068	0.050	0.015	0.080	0.044	0.069	1
	GLANCE-EA	0.084	0.072	0.159	0.135	0.183	0.069	0.106	0.283	0.136	0.097	0.102	0.098	0.104	0.105	0.125	0.171	0.122	0.098	0.078	1
	GLANCE-MD	0.096	0.084	0.138	0.130	0.161	0.089	0.104	0.141	0.137	0.119	0.115	0.098	0.121	0.108	0.111	0.141	0.126	0.110	0.096	0
	LineDP	0.099	0.052	0.123	0.090	0.060	0.157	0.068	0.081	0.070	0.090	0.069	0.055	0.030	0.071	0.055	0.018	0.071	0.041	0.081	1
	DeepLineDP	0.184	0.220	0.189	0.216	0.197	0.176	0.155	0.252	0.175	0.247	0.226	0.252	0.212	0.160	0.259	0.229	0.266	0.204	0.224	0
	N-gram	0.214	0.187	0.182	0.171	0.150	0.099	0.178	0.177	0.221	0.141	0.153	0.202	0.172	0.169	0.225	0.167	0.185	0.170	0.252	0
	ErrorProne	0.154	0.181	0.192	0.151	0.160	0.241	0.149	0.209	0.167	0.217	0.186	0.226	0.270	0.177	0.256	0.161	0.167	0.124	0.224	0
	Barinel	0.067	0.012	0.133	0.015	0.008	0.129	0.036	0.007	0.057	0.010	0.007	0.008	0.003	0.007	0.004	0.007	0.012	0.007	0.018	16
	Dstar	0.096	0.039	0.157	0.085	0.051	0.126	0.053	0.054	0.070	0.085	0.063	0.049	0.022	0.054	0.022	0.016	0.057	0.030	0.054	0
	Ochiai	0.085	0.025	0.156	0.047	0.035	0.134	0.042	0.034	0.062	0.054	0.032	0.027	0.006	0.036	0.007	0.010	0.033	0.019	0.030	0
	Op2	0.099	0.046	0.159	0.098	0.057	0.152	0.055	0.080	0.071	0.088	0.061	0.058	0.027	0.058	0.030	0.020	0.066	0.031	0.059	0
	Tarantula	0.077	0.023	0.138	0.038	0.029	0.132	0.041	0.030	0.064	0.042	0.024	0.027	0.009	0.029	0.010	0.011	0.030	0.017	0.026	0

TABLE IV: Comparison between our method With&Without File-Level Classifier. The gray background indicates better.

	Formula	With File-level		Without File-level	
		Mean	Median	Mean	Median
IFA	Barinel	13	0	15	0
	Dstar	59	8	101	15
	Ochiai	45	2	63	4
	Op2	69	10	200	28
	Tarantula	21	1	34	3
Recall@20%	Barinel	0.779	0.838	0.720	0.763
	Dstar	0.660	0.629	0.405	0.390
	Ochiai	0.724	0.744	0.554	0.532
	Op2	0.654	0.626	0.366	0.350
	Tarantula	0.749	0.796	0.604	0.605
Effort@20%	Barinel	0.019	0.007	0.018	0.007
	Dstar	0.052	0.044	0.086	0.091
	Ochiai	0.034	0.028	0.049	0.048
	Op2	0.057	0.048	0.102	0.110
	Tarantula	0.031	0.022	0.042	0.035

TABLE V: Comparison between SOUND With&Without Causality Analysis (CA). Gray background indicates better.

	Formula	With CA		Without CA	
		Mean	Median	Mean	Median
IFA	Barinel	13	0	13	0
	Dstar	59	8	64	8
	Ochiai	45	2	53	2
	Op2	69	10	79	13
	Tarantula	21	1	20	1
Effort@20%	Barinel	0.018661	0.006686	0.018713	0.006718
	Dstar	0.051530	0.043709	0.052014	0.043752
	Ochiai	0.034079	0.028403	0.034114	0.028403
	Op2	0.057117	0.047678	0.057656	0.049205
	Tarantula	0.030995	0.021898	0.031244	0.022304

We refine the original method by excluding the causality analysis (see Section III-B) and directly utilize the initial ranking results as the outcomes. This RQ aims to compare the performance with and without using the result of causality analysis to check its impact on our method.

Results. Table V displays the average and median values of the metrics. ① In terms of IFA, our methodology exhibits enhanced outcomes across four scenarios after utilizing causality analysis. The most notable improvement is a 23% ($\frac{13-10}{13}$) decrease in IFA. Conversely, in one scenario, there is an observed 5% ($\frac{21-20}{20}$) increment. ② Regarding Effort@20%, in 9/10 scenarios, our methods perform better, and in merely one scenario, the results reveal no distinguishable divergence up to the sixth decimal place. ③ Note that we do *not report* Recall@20% values here. When using causal tokens (see Step ③ in Section III-D), the defective ratio is based on the line-level defective rate from the prior release and is substantially lower than 20%, ensuring that with/without causal tokens does not impact on the Recall@20% values.

Answer to **RQ3**. The application of causality analysis markedly improves the IFA metric's performance and leads to a slight improvement under Effort@20%.

D. RQ4: Why Do Spectrum Information and Causality Analysis Work in CLDP?

Motivation&Approach. This RQ tries to explain *why* spectrum information analysis and causality analysis are beneficial for ranking defective lines in CR defect prediction.

(a) Considering our approach leveraging spectrum information to calculate the suspicion scores of tokens, we evaluate the consistency of the token suspicion ranking across different releases by computing the rank correlation of risky tokens.

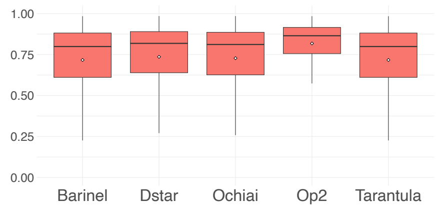


Fig. 5: Spearman correlation coefficient of the token ranking list under 5 formulas across 123 prediction release pairs

TABLE VI: Comparison of IFA On Effective and Non-effective set With and Without Causality Analysis (CA).

	Formula	With CA		Without CA	
		Mean	Median	Mean	Median
Non-Effective	Barinel	56	9	57	13
	Dstar	213	56	239	90
	Ochiai	175	22	212	67
	Op2	234	88	290	168
	Tarantula	82	16	79	18
Effective	Barinel	0	0	0	0
	Dstar	6	1	0	1
	Ochiai	1	0	0	0
	Op2	4	1	1	1
	Tarantula	0	0	0	0

We have chosen the Spearman correlation coefficient [53] as the metric for this evaluation. Our approach comprises the following three steps. ① We determine the shared tokens between the two releases through the intersection of sets. ② Token scores are extracted from both releases. ③ We calculate the Spearman rank correlation coefficient for the two score sequences using the `scipy.stats.spearmanr` [54] function.

(b) To comprehend the reasons behind the enhancement of IFA brought by causality analysis, based on prior research [55], we categorize 123 test releases according to the IFA results obtained `without` causality analysis as implemented in RQ3. We identify the top/bottom 25% releases as the **Effective set/Non-Effective set** with lower/higher IFA, respectively. An explanatory analysis is conducted by comparing the changes in IFA after utilizing causality analysis on the two distinct sets, which is recognized as effective.

Results. Figure 5 illustrates the Spearman correlation coefficients derived from the five formulas for calculating token suspicion. As shown in Figure 5, the Spearman correlation coefficients are relatively high, where the highest median is 0.865 under `Op2`. This observation indicates that token suspicion ranking based on spectrum information across different releases is relatively consistent, which could explain the advantage of employing spectrum information in CLDP.

Table VI shows that in the Non-Effective set, causality analysis leads to significant improvements in IFA across 9/10 scenarios, with the best case showing a 67% ($\frac{67-22}{67}$) enhancement. Regarding the Effective set, the performance of the releases adversely impacted the IFA in three scenarios, with the average IFA rising, though the maximum deterioration is limited to 6 (6-0) under `Dstar`. Conversely, the `Dstar`-based method's application to the Non-Effective set receives a significant improvement in the average IFA of 26 (239-213), notably overcoming the deterioration. We conclude that causality analysis demonstrates effectiveness through the notable IFA increase in the Non-Effective set.

TABLE VII: *Hit* and *Over* values between the Barinel-based method and the baseline methods on 19 stuided projects.

	LineDP		GLANCE-LR		DeepLineDP		ErrorProne		N-gram	
	Hit	Over	Hit	Over	Hit	Over	Hit	Over	Hit	Over
mean	94%	42%	93%	29%	82%	68%	80%	587%	80%	280%
median	96%	36%	98%	24%	90%	28%	90%	293%	86%	28%

Answer to **RQ4**. The effectiveness of spectrum information analysis is attributed to the high correlation in the token ranking between releases. Causality analysis demonstrates effectiveness through the notable increase in IFA for releases in the Non-Effective set.

VI. Discussion

This section provides a deeper discussion of our approach.

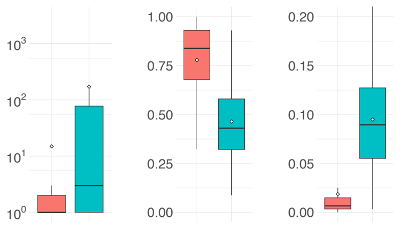
A. Can the proposed method replace the baseline methods?

This section aims to examine the classification difference between our methods and baselines from the perspective of true positive defective lines (i.e., the number of defective lines identified correctly). We choose the `Barinel`-based method as the representative since it shows the best performance (see RQ1). We compute two widely used metrics, Hit and $Over$, applied in related studies [11]: Hit represents the proportion of defective lines identified by baselines overlapped with those by our method; $Over$ represents the proportion of defective lines overlooked by baselines. Their calculation equations are as follows: $Hit = \frac{|TP_{\texttt{Barinel}} \cap TP_{\texttt{Baseline}}|}{|TP_{\texttt{Baseline}}|}$ and $Over = \frac{|TP_{\texttt{Barinel}} - TP_{\texttt{Baseline}}|}{|TP_{\texttt{Baseline}}|}$, where TP_m indicates the true positive defective lines identified by the method m.

Table VII shows that the `Barinel`-based method can identify more than 80% defective lines detected by the baselines at the mean, and in most cases more than 90% at the median. Considering *Over*, the `Barinel`-based method can identify many defective lines missed by baselines. For instance, 587% and 280% new defective lines missed by ErrorProne and N-gram are identified by our method at the mean. Meanwhile, 29%, 42%, and 68% new defective lines are found by our method compared to GLANCE-LR, LineDP, and DeepLineDP. In summary, the proposed method shows *significant potential to replace baseline methods and successfully identifies a proportion of defective lines that baselines fail to detect.*

B. Why We Choose Global Ranking?

In our proposed methods, we employ a global ranking strategy for all lines of code in defective files identified by the file-level classifier (see Section IV-B). In contrast, baseline methods typically rank defect files based on their suspicion scores and then rank the lines within each file based on their line-level suspicion. We call it File-level First, Line-level Later (FFLL) ranking. This section shows the performance improvement brought by the global ranking strategy across 123 target releases.



(a) IFA (↓) (b) Recall@20% (↑) (c) Effort@20% (↓)

Fig. 6: Comparison of Performance between Global Ranking marked in red and FFLL Ranking marked in green

Figure 6 illustrates the distribution of the IFA, Recall@20%, and Effort@20% metrics for the ranking results based on the `Barinel` suspicion calculation formula. Figure 6 shows, by employing a global ranking strategy, a significant improvement in the IFA at the mean from 170 to 13, and a substantial rise in Recall@20% from 0.464 to 0.779, and in Effort@20% from 0.095 to 0.019. In summary, the global ranking strategy demonstrates *a notable improvement in the method's ranking performance at the top of the list.*

C. How SOUND performs under other prediction settings?

In the previous experiment, we used the data from the **SPR** (Single Prior Release) for training. In this section, we compare our method with baselines under another prediction setting[8], the **APR** (All Prior Releases) scenario, i.e., for release i in the studied projects, we employ all previous releases, releases $1, \ldots, i-1$, as the training set to construct the models. We have conducted the same experiment under **APR** scenarios and compared the performance with our method and baselines.

Figure 7 shows that under the **APR** scenario, our methods perform better than baselines: for IFA, top 3 are our methods; for Recall@20%, the top 3 are our methods; and for Effort@20%, the top 5 are all our methods. In summary, *our methods also outperform baselines under APR scenarios.*

D. How SOUND performs under classification metrics?

We aim to explore the performance of our method under three classification metrics[9]: **AUC**, **D2H**, and **FAR**. Since 20%LOC is used in our study as a cutoff point for ranking indicators, we continue to employ 20%LOC as the threshold for classification. Our experiment first ranks all lines in a release in descending order based on predicted values. Then, the **top** 20% lines in the ranking list are labeled as defective, while the rest are categorized as clean.

Figure 8 shows the performance of our method and baselines under three classification indicators. As shown in Figure 8, the top 3 methods are always our methods under each classification metric. In summary, *our methods show better performance in classification tasks.*

[8]We have also conducted experiments under the Moving Window scenario, i.e., for release i, we employ k ($k = 3$) previous releases as the training set. We present the results in the online appendix (see Section IX-A).

[9]Due to space limitation, we present the results under another two classification indicators, F1-score and MCC, in the online appendix (see Section IX-A).

E. How SOUND performs under other inspection thresholds

In this section, we perform experiments to evaluate Recall@X%LOC (the percentage of total defective lines found within the first X% of code lines in a given release), where X is from 10 to 90 in intervals of 10.

Figure 9 shows the line plots of the medians of Recall@X%LOC for our method and the baseline methods among prediction results of 123 release pairs. The horizontal axis represents X%, while the vertical axis corresponds to the Recall@X%LOC value. In Figure 9, our three methods (`Barinel`, `Tarantula`, and `Ochiai`) outperform the baselines under Recall10%LOC, Recall20%LOC, and Recall30%LOC. In summary, *our methods outperform baselines under other effort-aware metrics like Recall10%LOC and Recall30%LOC.*

F. Insights of our study

Our method utilizes spectrum information to quantify the contribution of tokens to line-level defects and causality analysis to demonstrate a deeper causal relationship between the tokens and defects. The experimental results demonstrate that our method can enhance the ranking performance of CLDP. Our study may lead to insights for the following studies:

(a) Making comprehensive use of historical line-level label information can effectively enhance the ranking performance. Line-level defect prediction techniques necessitate a comprehensive consideration of the contribution of individual tokens to the defect of a code line.

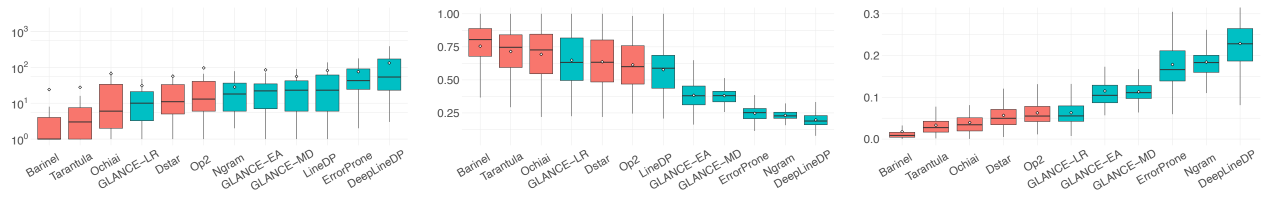
(b) Hybrid methods, e.g., with program spectrum perspective and causality analysis, have been proven to be effective in addressing the issue of line-level defect prediction. Employing diverse perspectives allows us to holistically contemplate the impact of tokens on defect occurrences.

VII. Thread to Validity

This section discusses the threats to the validity of our approach. ① The quantity and comprehensiveness of the dataset significantly affect the study's validity. We select 19 open-source datasets commonly used in defect prediction research to minimize biases inherent in smaller datasets. Future studies should consider testing on larger and more diverse datasets to ensure the method's applicability across different projects. ② We have employed five spectrum-based formulas and the causality analysis tool `DiBS`, both considered effective according to research studies, also confirmed by our experiments. In the future, there is still scope for exploring even more effective tools better suited to defect prediction scenarios. ③ The selection of baselines may compromise the validity of the experiment. The methods chosen in this paper are derived from top-tier international conferences and journals. This paper discloses the details of the code. However, due to the subjectivity inherent in code implementation, further verification should be conducted in future research.

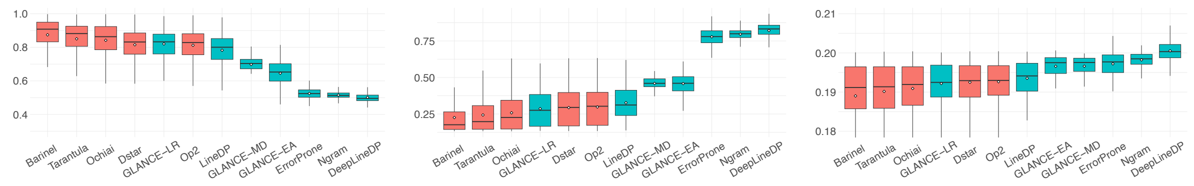
VIII. Related Work

Line-level Defect Prediction. FindBugs [56], PMD [21] and CheckStyle [22] are commonly used static analysis tools that identify defect lines according to predefined rules. Majd et



(a) IFA (↓) boxplots in the increasing order (b) Recall@20%(↑)boxplots in the decreasing order (c) Effort@20% (↓) boxplots in the increasing order

Fig. 7: The Boxplots of our methods marked red and baseline methods marked green under **APR** scenario, where ↑/↓ indicates that a higher/lower value of the metric is better.



(a) AUC (↑) boxplots in the decreasing order (b) D2H (↓) boxplots in the increasing order (c) FAR (↓) boxplots in the increasing order

Fig. 8: The Boxplots of **classification indicator** values of our methods marked red and baseline methods marked green, where ↑/↓ indicates that a higher/lower value of the metric is better.

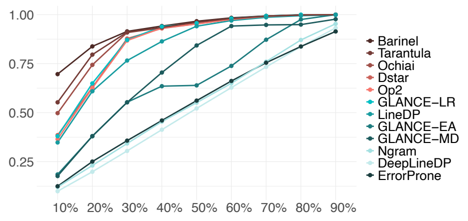


Fig. 9: Line plots of medians of Recall@X%LOC(↑) for our methods and baselines

al. [57] introduced SLDeep, which uses 32 statement-level metrics and the long-short-term memory (LSTM) learning model for the prediction of defects. The N-gram [10], [51], [52] language model is recommended to predict unnatural code phrases based on language statistics instead of rules. Wattanakriengkrai et al. [9] utilized LIME, a model-agnostic technique, for line-level defect prediction. Guo et al. [11] introduced GLANCE, focusing on control elements, function calls, and the count of tokens per line. Pornprasit et al. [12] presented DeepLineDP, a deep learning approach that uses attention mechanisms to quantify the suspiciousness of tokens. *In contrast to these methods that focus only on the textual content and the file-level impact of tokens on line-level defect prediction, our approach fully leverages line-level label information, employing a concise method to quantify the contribution of tokens to defective lines.*

Program Spectrum. Renieris and Reiss [58] introduced a fault localization technique, comparing the spectrum of faulty and the closest correct runs by a distance metric. Jones [39] introduced Tarantula, based on ESHS similarity coefficients, using coverage and execution outcomes (pass/fail). Abreu et al. [37] proposed Ochiai and considered it more effective than Tarantula. Wong et al. proposed Dstar [36], a technique for automated fault localization without requiring prior program structure or semantic information. As proven by Yoo et al. [59], there is no universally best spectrum-based technique.

Causality Analysis. Johnson et al. [60] proposed a novel root cause analysis method called CausalTesting, which uses counterfactual causality theory to understand and fix bugs. Dubslaff et al. [61] introduced the concept of causality in configurable software systems. He et al. [62] introduced PerfSig, a multi-modality tool for identifying performance bugs' primary anomaly patterns and root cause functions. Sun et al. [63] proposed CARE (CAusality-based REpair), a neural network repair method based on causality. Ji et al. [16], [64] introduced the first causality-aware DNN coverage criterion and employed causal analysis in ML pipelines to explore the trade-offs between fairness parameters and other key metrics.

IX. Conclusion

This paper proposes SOUND, a novel boosting method for CLDP. Unlike previous prediction methods, SOUND combines spectrum information and causal analysis to exploit historical line-level label information fully. Our experiments demonstrate that SOUND enhances the ranking performance at the top. In future research, we aim to employ more effective spectrum information formulas and causal analysis techniques to further improve the efficacy of our method.

A. Replication Package and Appendix

We present the dataset, source code files, and the Appendix with details online at https://github.com/shiyusunse/SOUND

X. Acknowledgments

The authors thank anonymous reviewers for their time and invaluable feedback to improve our study. This work is supported by the National Natural Science Foundation of China (Grant Nos. 62172202, 62272221, 62172205, and 61972082).

References

[1] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, “A systematic literature review on fault prediction performance in software engineering,” *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276–1304, 2011.

[2] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, “Revisiting code ownership and its relationship with software quality in the scope of modern code review,” in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 1039–1050.

[3] ——, “Investigating code review practices in defective files: An empirical study of the qt system,” in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 168–179.

[4] Ö. F. Arar and K. Ayan, “Software defect prediction using cost-sensitive neural network,” *Applied Soft Computing*, vol. 33, pp. 263–277, 2015.

[5] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan, “Revisiting common bug prediction findings using effort-aware models,” in *2010 IEEE international conference on software maintenance*. IEEE, 2010, pp. 1–10.

[6] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K.-i. Matsumoto, “The effects of over and under sampling on fault-prone module detection,” in *First international symposium on empirical software engineering and measurement (ESEM 2007)*. IEEE, 2007, pp. 196–204.

[7] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, “A large-scale empirical study of just-in-time quality assurance,” *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[8] T. Mende and R. Koschke, “Effort-aware defect prediction models,” in *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 107–116.

[9] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto, “Predicting defective lines using a model-agnostic technique,” *IEEE Trans. Software Eng.*, vol. 48, no. 5, pp. 1480–1496, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2020.3023177

[10] V. J. Hellendoorn and P. Devanbu, “Are deep neural networks the best choice for modeling source code?” in *Proceedings of the 2017 11th Joint meeting on foundations of software engineering*, 2017, pp. 763–773.

[11] Z. Guo, S. Liu, X. Liu, W. Lai, M. Ma, X. Zhang, C. Ni, Y. Yang, Y. Li, L. Chen, G. Zhou, and Y. Zhou, “Code-line-level bugginess identification: How far have we come, and how far have we yet to go?” *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, pp. 102:1–102:55, 2023. [Online]. Available: https://doi.org/10.1145/3582572

[12] C. Pornprasit and C. K. Tantithamthavorn, “Deeplinedp: Towards a deep learning approach for line-level defect prediction,” *IEEE Trans. Software Eng.*, vol. 49, no. 1, pp. 84–98, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2022.3144348

[13] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, “Evaluating and improving fault localization,” in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 609–620.

[14] H. Agrawal, R. A. De Millo, and E. H. Spafford, “An execution-backtracking approach to debugging,” *IEEE Software*, vol. 8, no. 3, pp. 21–26, 1991.

[15] T. Baluta, S. Shen, S. Hitarth, S. Tople, and P. Saxena, “Membership inference attacks and generalization: A causal perspective,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 249–262.

[16] Z. Ji, P. Ma, S. Wang, and Y. Li, “Causality-aided trade-off analysis for machine learning fairness,” in *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 2023, pp. 371–383. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00105

[17] Q. Huang, X. Xia, and D. Lo, “Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction,” in *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 2017, pp. 159–170. [Online]. Available: https://doi.org/10.1109/ICSME.2017.51

[18] Z. Yu, F. M. Fahid, H. Tu, and T. Menzies, “Identifying self-admitted technical debts with jitterbug: A two-step approach,” *IEEE Trans. Software Eng.*, vol. 48, no. 5, pp. 1676–1691, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2020.3031401

[19] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan, “Building useful program analysis tools using an extensible java compiler,” in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2012, pp. 14–23.

[20] H. Hata, O. Mizuno, and T. Kikuno, “Bug prediction based on fine-grained module histories,” in *2012 34th international conference on software engineering (ICSE)*. IEEE, 2012, pp. 200–210.

[21] A. Trautsch, S. Herbold, and J. Grabowski, “A longitudinal study of static analysis warning evolution and the effects of PMD on software quality in apache open source projects,” in *Software Engineering 2021, Fachtagung des GI-Fachbereichs Softwaretechnik, 22.-26. Februar 2021, Braunschweig/Virtuell*, ser. LNI, A. Koziolek, I. Schaefer, and C. Seidl, Eds., vol. P-310. Gesellschaft für Informatik e.V., 2021, pp. 107–108. [Online]. Available: https://doi.org/10.18420/SE2021_41

[22] ——, “Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction,” in *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 2020, pp. 127–138. [Online]. Available: https://doi.org/10.1109/ICSME46990.2020.00022

[23] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[24] J. S. Collofello and L. Cousins, “Towards automatic software fault location through decision-to-decision path analysis,” in *Managing Requirements Knowledge, International Workshop on*. IEEE Computer Society, 1987, pp. 539–539.

[25] B. Korel, “PELAS - program error-locating assistant system,” *IEEE Trans. Software Eng.*, vol. 14, no. 9, pp. 1253–1260, 1988. [Online]. Available: https://doi.org/10.1109/32.6169

[26] B. Korel and J. Laski, “Stad-a system for testing and debugging: User perspective,” in *Workshop on Software Testing, Verification, and Analysis*. IEEE Computer Society, 1988, pp. 13–14.

[27] A.-B. Taha, S. M. Thebaut, and S.-S. Liu, “An approach to software fault localization and revalidation based on incremental data flow analysis,” in *[1989] Proceedings of the Thirteenth Annual International Computer Software & Applications Conference*. IEEE, 1989, pp. 527–534.

[28] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, “Fault localization using execution slices and dataflow tests,” in *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*. IEEE, 1995, pp. 143–151.

[29] J. A. Jones and M. J. Harrold, “Empirical evaluation of the tarantula automatic fault-localization technique,” in *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, D. F. Redmiles, T. Ellman, and A. Zisman, Eds. ACM, 2005, pp. 273–282. [Online]. Available: https://doi.org/10.1145/1101908.1101949

[30] W. E. Wong, V. Debroy, and B. Choi, “A family of code coverage-based heuristics for effective fault localization,” *Journal of Systems and Software*, vol. 83, no. 2, pp. 188–208, 2010.

[31] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, “Evaluating and improving fault localization,” in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE / ACM, 2017, pp. 609–620. [Online]. Available: https://doi.org/10.1109/ICSE.2017.62

[32] J. Pearl, *Causality*. Cambridge university press, 2009.

[33] L. Lorch, J. Rothfuss, B. Schölkopf, and A. Krause, “Dibs: Differentiable bayesian structure learning,” *Advances in Neural Information Processing Systems*, vol. 34, pp. 24 111–24 123, 2021.

[34] “Countvectorizer,” https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html.

[35] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, “Spectrum-based multiple fault localization,” in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 88–99.

[36] W. E. Wong, V. Debroy, R. Gao, and Y. Li, “The dstar method for effective software fault localization,” *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2013.

[37] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, “An evaluation of similarity coefficients for software fault localization,” in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 2006, pp. 39–46.

[38] L. Naish, H. J. Lee, and K. Ramamohanarao, “A model for spectra-based software diagnosis,” *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, pp. 1–32, 2011.

[39] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization for fault localization," in *in Proceedings of ICSE 2001 Workshop on Software Visualization*. Citeseer, 2001.

[40] Z. Ji, P. Ma, Z. Li, and S. Wang, "Benchmarking and explaining large language model-based code generation: A causality-centric approach," *CoRR*, vol. abs/2310.06680, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2310.06680

[41] "dibs.models package," https://differentiable-bayesian-structure-learning.readthedocs.io/en/latest/items/models.html.

[42] Y. Liu, Y. Li, J. Guo, Y. Zhou, and B. Xu, "Connecting software metrics across versions to predict defects," in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, R. Oliveto, M. D. Penta, and D. C. Shepherd, Eds. IEEE Computer Society, 2018, pp. 232–243. [Online]. Available: https://doi.org/10.1109/SANER.2018.8330212

[43] M. J. Ahmad, K. Goseva-Popstojanova, and R. R. Lutz, "The untold impact of learning approaches on software fault-proneness predictions: an analysis of temporal aspects," *Empirical Software Engineering*, vol. 29, no. 4, p. 87, 2024.

[44] S. Tunkel and S. Herbold, "Exploring the relationship between performance metrics and cost saving potential of defect prediction models," *Empirical Software Engineering*, vol. 27, no. 7, p. 182, 2022.

[45] X. Yu, J. Rao, L. Liu, G. Lin, W. Hu, J. W. Keung, J. Zhou, and J. Xiang, "Improving effort-aware defect prediction by directly learning to rank software modules," *Information and Software Technology*, vol. 165, p. 107250, 2024.

[46] C. Zhou, P. He, C. Zeng, and J. Ma, "Software defect prediction with semantic and structural information of codes based on graph neural networks," *Information and Software Technology*, vol. 152, p. 107057, 2022.

[47] S. Amasaki, "Cross-version defect prediction: use historical data, cross-project data, or both?" *Empir. Softw. Eng.*, vol. 25, no. 2, pp. 1573–1595, 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09777-8

[48] M. A. Kabir, J. W. Keung, K. E. Bennin, and M. Zhang, "Assessing the significant impact of concept drift in software defect prediction," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2019, pp. 53–58.

[49] A. K. Gangwar and S. Kumar, "Concept drift in software defect prediction: a method for detecting and handling the drift," *ACM Transactions on Internet Technology*, vol. 23, no. 2, pp. 1–28, 2023.

[50] C. Pornprasit and C. K. Tantithamthavorn, "Deeplinedp: Towards a deep learning approach for line-level defect prediction," *IEEE Trans. Software Eng.*, vol. 49, no. 1, pp. 84–98, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2022.3144348

[51] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the" naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 428–439.

[52] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: bug detection with n-gram language models," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 708–719.

[53] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, pp. 531–577, 2012.

[54] "spearmanr — scipy v1.14.0 manual," https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.spearmanr.html

[55] G. Grano, F. Palomba, and H. C. Gall, "Lightweight assessment of test-case effectiveness using source-code-quality indicators," *IEEE Transactions on Software Engineering*, vol. 47, no. 4, pp. 758–774, 2019.

[56] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan, "Building useful program analysis tools using an extensible java compiler," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2012, pp. 14–23.

[57] A. Majd, M. Vahidi-Asl, A. Khalilian, P. Poorsarvi-Tehrani, and H. Haghighi, "Sldeep: Statement-level software defect prediction using deep-learning model on static code features," *Expert Systems with Applications*, vol. 147, p. 113156, 2020.

[58] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 30–39.

[59] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman, "No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist," *RN*, vol. 14, no. 14, p. 14, 2014.

[60] B. Johnson, Y. Brun, and A. Meliou, "Causal testing: understanding defects' root causes," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 87–99. [Online]. Available: https://doi.org/10.1145/3377811.3380377

[61] C. Dubslaff, K. Weis, C. Baier, and S. Apel, "Causality in configurable software systems," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 325–337. [Online]. Available: https://doi.org/10.1145/3510003.3510200

[62] J. He, Y. Lin, X. Gu, C. M. Yeh, and Z. Zhuang, "Perfsig: Extracting performance bug signatures via multi-modality causal analysis," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1669–1680. [Online]. Available: https://doi.org/10.1145/3510003.3510110

[63] B. Sun, J. Sun, L. H. Pham, and J. Shi, "Causality-based neural network repair," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 338–349.

[64] Z. Ji, P. Ma, Y. Yuan, and S. Wang, "CC: causality-aware coverage criterion for deep neural networks," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1788–1800. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00153