Predicting Defective Lines Using a Model-Agnostic Technique

Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto

Abstract—Defect prediction models are proposed to help a team prioritize source code areas files that need Software Quality Assurance (SQA) based on the likelihood of having defects. However, developers may waste their unnecessary effort on the whole file while only a small fraction of its source code lines are defective. Indeed, we find that as little as 1%-3% of lines of a file are defective. Hence, in this work, we propose a novel framework (called LINE-DP) to identify defective lines using a model-agnostic technique, i.e., an Explainable AI technique that provides information why the model makes such a prediction. Broadly speaking, our LINE-DP first builds a file-level defect model using code token features. Then, our LINE-DP uses a state-of-the-art model-agnostic technique (i.e., LIME) to identify risky tokens, i.e., code tokens that lead the file-level defect model to predict that the file will be defective. Then, the lines that contain risky tokens are predicted as defective lines. Through a case study of 32 releases of nine Java open source systems, our evaluation results show that our LINE-DP achieves an average recall of 0.61, a false alarm rate of 0.47, a top 20%LOC recall of 0.27, and an initial false alarm of 16, which are statistically better than six baseline approaches. Our evaluation shows that our LINE-DP requires an average computation time of 10 seconds including model construction and defective identification time. In addition, we find that 63% of defective lines that can be identified by our LINE-DP are related to common defects (e.g., argument change, condition change). These results suggest that our LINE-DP can effectively identify defective lines that contain common defects while requiring a smaller amount of inspection effort and a manageable computation cost. The contribution of this paper builds an important step towards line-level defect prediction by leveraging a model-agnostic technique.

Index Terms—Software Quality Assurance, Line-level Defect Prediction

1 INTRODUCTION

Software Quality Assurance (SQA) is one of software engineering practices for ensuring the quality of a software product [26]. When changed files from the cutting-edge development branches will be merged into the release branch where the quality is strictly controlled, an SQA team needs to carefully analyze and identify software defects in those changed files [1]. However, due to the limited SQA resources, it is infeasible to examine the entire changed files. Hence, to spend the optimal effort on the SQA activities, an SQA team needs to prioritize files that are likely to have defects in the future (e.g., post-release defects).

Defect prediction models are proposed to help SQA teams prioritize their effort by analyzing post-release software defects that occur in the previous release [16, 26, 55, 59, 77, 80]. Particularly, defect prediction models leverage the information extracted from a software system using product metrics, the development history using process metrics, and textual content of source code tokens. Then, the defect models estimate defect-proneness, i.e., the likelihood that a file will be defective after a software product is released. Finally, the files are prioritized based on the defect-proneness.

• *S. Wattanakriengkrai, H. Hata, and K. Matsumoto are with Nara Institute of Science and Technology, Japan.*
E-mail: {wattanakri.supatsara.ws3, hata, matumoto}@is.naist.jp.
• *P. Thongtanunam is with the University of Melbourne, Australia.*
E-mail: patanamon.t@unimelb.edu.au.
• *C. Tantithamthavorn is with Monash University, Australia.*
E-mail: chakkrit@monash.edu.

To achieve effective SQA prioritization, defect prediction models have been long investigated at different granularity levels, for example, packages [43], components [82], modules [44], files [43, 53], methods [28], and commits [45]. However, developers could still waste an SQA effort on manually identifying the most risky lines, since the current prediction granularity is still perceived as coarse-grained [87]. In addition, our motivating analysis shows that as little as 1%-3% of the lines of code in a file are actually defective after release, suggesting that developers could waste their SQA effort on up to 99% of clean lines of a defective file. Thus, line-level defect prediction models would ideally help the team to save a huge amount of the SQA effort.

In this paper, we propose a novel line-level defect prediction framework which leverages a model-agnostic technique (called LINE-DP) to predict defective lines, i.e., the source code lines that will be changed by bug-fixing commits to fix post-release defects. Broadly speaking, our LINE-DP will first build a file-level defect model using code token features. Then, our LINE-DP uses a state-of-the-art model-agnostic technique (i.e., LIME [69]) to explain a prediction of which code tokens lead the file-level defect model to predict that the file will be defective. Finally, the lines that contain those code tokens are predicted as defective lines. The intuition behind our approach is that *code tokens that frequently appeared in defective files in the past may also appear in the lines that will be fixed after release*.

In this work, we evaluate our LINE-DP in terms of (1) predictive accuracy, (2) ranking performance, (3) computation time, and (4) the types of uncovered defects. We also compare our LINE-DP against six baseline approaches that are potential to identify defective lines based on the literature, i.e., random guessing, a natural language processing (NLP) based approach, two static analysis tools (i.e., Google's ErrorProne and PMD), and two traditional model interpretation (TMI) based approaches using logistic regression and random forest. The evaluation is based on both within-release and cross-release validation settings. Through a case study of 32 releases of 9 software systems, our empirical evaluation shows that our LINE-DP achieves an average recall of 0.61, a false alarm rate of 0.47, a top 20%LOC recall of 0.27, and an initial false alarm of 16 which are significantly better than the baseline approaches. The average computation time (including the model construction and line identification time) of our LINE-DP is 10.68 and 8.46 seconds for the within-release and cross-release settings, respectively. We find that 63% of the defective lines identified by our LINE-DP are categorized into the common defect types. Our results lead us to conclude that leveraging a model-agnostic technique can effectively identify and rank defective lines that contain common defects while requiring a manageable computation cost. Our work builds an important step towards line-level defect prediction by leveraging a model-agnostic technique.

Novelty Statement. To the best of our knowledge, our work is the first to use the machine learning-based defect prediction models to predict defective lines by leveraging a model-agnostic technique from the Explainable AI domain. More specifically, this paper is the first to present:

- A novel framework for identifying defective lines that uses a state-of-the-art model-agnostic technique.
- An analysis of the prevalence of defective lines.
- The benchmark line-level defect datasets are available online at https://github.com/awsm-research/line-level-defect-prediction.
- A comprehensive evaluation of line-level defect prediction in terms of predictive accuracy (RQ1), ranking performance (RQ2), computation cost (RQ3), and the types of uncovered defects (RQ4).
- A comparative evaluation between our framework and six baseline approaches for both within-release and cross-release evaluation settings.

Paper Organization. The rest of our paper is organized as follows: Section 2 introduces background of software quality assurance and defect prediction models. Section 3 presents a motivating analysis. Section 4 discusses the related work. Section 5 describes our framework. Section 6 describes the design of our experiment. Section 7 presents the results of our experiment. Section 8 discusses the limitation and discloses the potential threats to validity. Section 9 draws the conclusions.

2 BACKGROUND

In this section, we provide background of software quality assurance and defect prediction models.

2.1 Software Quality Assurance

Software Quality Assurance (SQA) is a software engineering practice to ensure that a software product meets the quality standards, especially for the life-impacting and safety-critical software systems. Thus, SQA practices must be

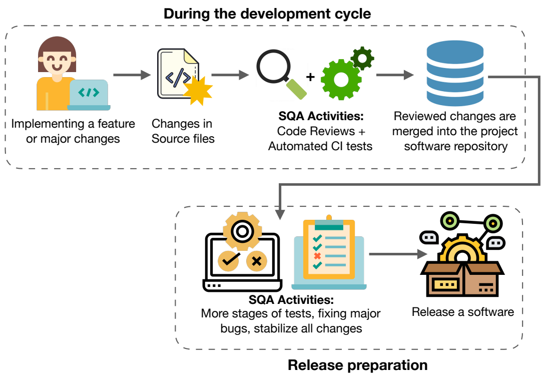

During the development cycle
Implementing a feature or major changes
Changes in Source files
SQA Activities: Code Reviews + Automated CI tests
Reviewed changes are merged into the project software repository
SQA Activities: More stages of tests, fixing major bugs, stabilize all changes
Release a software
Release preparation

Fig. 1: An overview of SQA activities in the Software Engineering workflow [1].

embedded as a quality culture throughout the life cycles from planning, development stage, to release preparation so teams can follow the best practices to prevent software defects. Figure 1 illustrates a simplified software engineering workflow that includes SQA activities [1].

2.1.1 SQA activities during the development stage

During the development stage, new features and other code changes are implemented by developers. Such code changes (or commits) must undergo rigorous SQA activities (e.g., Continuous Integration tests and code review) prior to merge into the main branch (e.g., a master branch) [24]. Since these commit-level SQA activities are time-consuming, Just-In-Time defect prediction has been proposed to support developers by prioritizing their limited SQA effort on the most risky code changes that will introduce software defects during the development cycle (i.e., pre-release defects) [45, 61]. Nevertheless, JIT defect prediction only early detects defect-inducing changes, rather than post-release defects (i.e., the areas of code that are likely to be defective after a release). Despite the SQA activities during the development cycle (e.g., code reviews), it is still possible that software defects still slip through to the official release of software products [81, 82]. Thus, SQA activities are still needed during the release preparation.

2.1.2 SQA activities during the release preparation

During the release preparation, intensive SQA activities must be performed to ensure that the software product is of high quality and is ready for release, i.e., reducing the likelihood that a software product will have post-release defects [1, 58]. In other words, the files that are changed during the software development need to be checked and stabilized to ensure that these changes will not impact the overall quality of the software systems [32, 50, 65]. Hence, several SQA activities (e.g., regression tests, manual tests) are performed [1]. However, given thousands of files that need to be checked and stabilized before release, it is intuitively infeasible to exhaustively perform SQA activities for all of the files of the codebase with the limited SQA resources (e.g., time constraints), especially in rapid-release development practices. To help practitioners effectively prioritize their limited SQA resources, it is of importance to identify **what are the most defect-prone areas of source code that are likely to have post-release defects**.

Prior work also argued that it is beneficial to obtain early estimates of defect-proneness for areas of source code to help software development teams develop the most effective SQA resource management [57, 58]. Menzies *et al.* mentioned that software contractors tend to prioritize their effort on reviewing software modules tend to be fault-prone [55]. A case study at ST-Ericsson in Lund, Sweden by Engström *et al.* [17] found that the selection of regression test cases guided by the defect-proneness of files is more efficient than the manual selection approaches. At the Tizen-wearable project by Samsung Electronics [46], they found that prioritizing APIs based on their defect-proneness increases the number of discovered defects and reduces the cost required for executing test cases.

2.2 Defect Prediction Models

Defect prediction models have been proposed to predict the most risky areas of source code that are likely to have post-release defects [16, 55, 59, 77, 80, 89, 90]. A defect prediction model is a classification model that estimates the likelihood that a file will have post-release defects. One of the main purposes is to help practitioners effectively spend their limited SQA resources on the most risky areas of code in a cost-effective manner.

2.2.1 The modelling pipeline of defect prediction models

The predictive accuracy of the defect prediction model heavily relies on the modelling pipelines of defect prediction models [4, 22, 56, 73, 74, 76, 78]. To accurately predicting defective areas of code, prior studies conducted a comprehensive evaluation to identify the best technique of the modelling pipelines for defect models. For example, feature selection techniques [23, 39, 40], collinearity analysis [37–39], class rebalancing techniques [75], classification techniques [22], parameter optimization [4, 21, 77, 80], model validation [79], and model interpretation [36, 37]. Despite the recent advances in the modelling pipelines for defect prediction models, the cost-effectiveness of the SQA resource prioritization still relies on the granularity of the predictions.

2.2.2 The granularity levels of defect predictions models

The cost-effectiveness of the SQA resource prioritization heavily relies on the granularity levels of defect prediction. Prior studies argued that prioritizing software modules at the finer granularity is more cost-effective [28, 43, 61]. For example, Kamei *et al.* [43] found that the file-level defect prediction is more effective than the package-level defect prediction. Hata *et al.* [28] found that the method-level defect prediction is more effective than file-level defect prediction. Defect models at various granularity levels have been proposed, e.g., packages [43], components [82], modules [44], files [43, 53], methods [28]. However, developers could still waste an SQA effort on manually identifying the most risky lines, since the current prediction granularity is still perceived as coarse-grained [87]. Hence, the line-level defect prediction should be beneficial to SQA teams to spend optimal effort on identifying and analyzing defects.

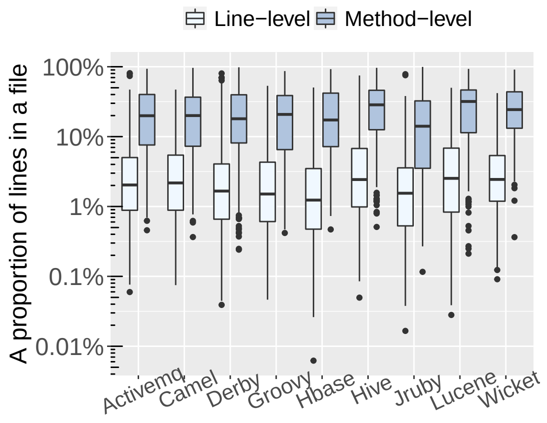


Fig. 2: The proportion of lines in a file that are inspected when using the line-level and method-level defect prediction models.

3 Motivating Analysis

In this section, we perform a quantitative analysis in order to better understand how much SQA effort could be spent when defect-proneness is estimated at different granularities. It is possible that developers may waste their SQA effort on a whole file (or a whole method) while only a small fraction of its source code lines are defective.

An illustrative example. Given that a defective file f has a total lines of 100, all 100 lines in the file will require SQA effort if the defect-proneness is estimated at a file level. However, if the defect-proneness is estimated at a method level, only lines in a defective method m (i.e., the method that contains defective lines) will require SQA effort. Assuming that this defective method has 30 lines, the required SQA effort will be 30% of the effort required at a file level. If the defect-proneness is estimated at a line level, only defective lines will require SQA effort. Assuming that there are 5 defective lines in this file, the required SQA effort will be only 5% of the effort required at a file level.

Approach. To quantify possible SQA effort when the defect-proneness is estimated at the file, method, or line levels, we measure the proportion of defect-prone lines. To do so, we first extract defective lines, i.e., the lines in the released system that were removed by bug-fixing commits after release (see Section 6.2).[1] Then, for each defective file, we measure the proportion of defect-prone lines at the line level, i.e., $\frac{\#\text{DefectiveLines}}{\text{LOC}_f}$, where LOC_f is the total number of lines in a defective file f. We also identify defective methods, i.e., methods in the defective file f that contain at least one defective line. Then, we measure the proportion of defect-prone lines at the method level, i.e., $\frac{\sum_{m \in M} \text{LOC}_\text{m}}{\text{LOC}_f}$, where LOC_m is the number of lines in a defective method m and M is a set of defective methods in the defective file f. Finally, we examine the distributions of the proportion of defective lines across the 32 studied releases of the nine studied systems.

1. Note that in this analysis, we only focus on the defective lines in the defective files, i.e., the files that are only impacted by bug-fixing commits.

Results. We find that as little as 1.2% - 2.5% of the lines in a defective file are defective. Figure 2 shows the distributions of the proportion of defect-prone lines in a defective file. We find that at the median, 1.2% to 2.5% of lines in the defective files are defective lines, i.e., the lines that actually impacted by bug-fixing commits. Moreover, we observe that 21% (Hive) - 46% (Camel) of defective files have only one single defective line. As we suspected, this result indicates that only a small fraction of source code lines in the defective files are defective. This suggests that when using file-level defect prediction, developers could unnecessarily spend their SQA effort on 97% - 99% of clean lines in a defective file.

Furthermore, Figure 2 presents the distributions of the proportion of defect-prone lines when using method-level prediction. At the median, the defective methods account for 14% - 32% of lines in a defective file. This suggests that in our studied releases, the proportion of defect-prone lines predicted at the method level is still relatively larger than those defect-prone lines predicted at the line level. Hence, *a more fine-grained approach to predict and prioritize defective lines could substantially help developers to reduce their SQA effort*.

4 Related Work

In this section, we discuss the state-of-the-art techniques that identify defect-prone lines, i.e., static analysis approaches and NLP-based approaches. We also discuss the challenges when using machine learning to build line-level defect prediction models.

4.1 Static Analysis

Static analysis is a tool that checks source code and reports warnings (i.e., common errors such as null pointer dereferencing and buffer overflows) at the line level. Various static analysis approaches are proposed including heuristic rule-based techniques (e.g., PMD [12]), complex algorithms [20], and hybrid approaches like FindBugs[2] which incorporates the static data-flow analysis and the pattern-matching analysis. A static analysis tool could potentially be used to predict and rank defect-prone lines [85]. However, Kremenek *et al.* [48] argued that Static Bug Finder (SBF) often reported false warnings, which could waste developers' effort. Several studies proposed approaches to filter and prioritize warnings reported by SBF [29, 30, 48, 49, 70]. Recently, Rahman *et al.* [64] found that the warnings reported by a static analysis tool can be used to prioritize defect-prone files. However, they found that their studied static analysis tools (i.e., PMD and FindBugs) and the file-level defect prediction models provide comparable benefits, i.e., the ranking performance between the defect models and static analysis tools is not statistically different. Yet, little has is known about whether the line-level defect prediction is better than a static analysis or not.

4.2 NLP-based Approaches

With a concept of software naturalness, statistical language models from Natural Language Processing (NLP) have been used to measure the repetitiveness of source code in a software repository [33]. Prior work found that statistical language models can be leveraged to help developers in many software engineering tasks such as code completion [68, 84], code convention [5], and method names suggestion [6]. Generally speaking, language models statistically estimate the probability that a word (or a code token) in a sentence (or a source code line) will follow previous words. Instead of considering all previous words in a sentence, one can use **n-gram** language models which use Markov assumptions to estimate the probability based on the preceding $n-1$ words. Since the probabilities may vary by the orders of magnitude, **entropy** is used to measure the naturalness of a word while considering the probabilities of the proceeding words. In other words, entropy is a measure of how surprised a model is by the given word. An entropy value indicates the degree that a word is *unnatural* in a given context (i.e., the preceding $n-1$ words).

Recent work leverages the n-gram language models to predict defect-prone tokens and lines [67, 88]. More specifically, Wang *et al.* [88] proposed an approach (called Bugram) which identifies the defective code tokens based on the probabilities estimated by n-gram models. To evaluate Bugram, Wang *et al.* manually examined whether the predicted tokens are considered as true defects based on specific criteria such as incorrect project specific function calls and API usage violation. On the other hand, Ray *et al.* [67] examined the naturalness of defective lines (i.e., lines that are removed by bug-fixing commits) based on the entropy of probabilities that are estimated by n-gram models. Ray *et al.* also found that ranking the files based on an average entropy of lines is comparable to ranking source files based on the probability estimated by the file-level defect prediction models. However, little is known about whether ranking defect-prone lines based on entropy is better than a line-level defect prediction model or not.

4.3 Challenges in Machine Learning-based Approaches

The key challenge of building traditional defect models at the line level is the design of hand-crafted software metrics. The state-of-the-art software metrics (e.g., code and process metrics) are often calculated at the class, file, and method levels [28, 63]. Extracting those features at the line level is not a trivial task since one would need to acquire accurate historical data for each line in the source code files. In the literature, the finest-grained defect prediction models are currently at the method level [28].

Instead of using hand-crafted software metrics, prior work directly uses semantic features of source code to build defect prediction models [7, 14, 35, 89, 90]. For example, Wang *et al.* [89] automatically generate semantic features from source code using a deep belief network and train a file-level defect prediction model using traditional classification techniques (e.g., Logistic Regression). Despite the success of using semantic features for file-level defect prediction, the size of the training datasets is still highly-dimensional and sparse (i.e., there is a large number of tokens and a large number of files). Given a huge amount of source code lines (e.g., 75K+ lines), it is likely infeasible

2. http://findbugs.sourceforge.net/

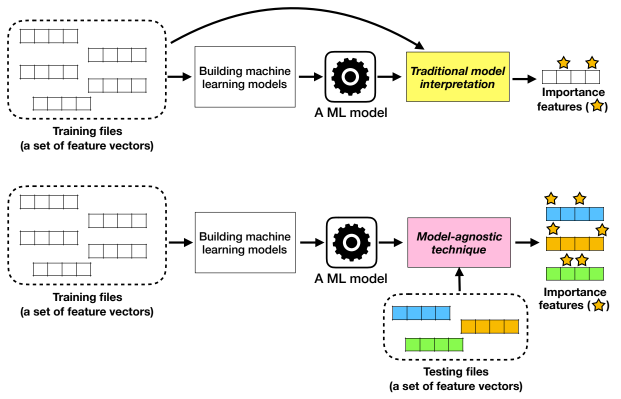

Training files
(a set of feature vectors)
Building machine learning models
A ML model
Traditional model interpretation
Importance features
Model-agnostic technique
Testing files
(a set of feature vectors)

Fig. 3: An illustrative comparison between traditional model interpretation and model-agnostic techniques.

and impractical to build a line-level defect prediction model using semantic features within a reasonable time. To demonstrate this, we built a line-level defect prediction model using the smallest defect dataset (i.e., 230,898 code tokens and 259,617 lines of code) with the simplest ML learning algorithm (i.e., Logistic Regression) with semantic features (e.g., the frequency of code tokens). Our preliminary analysis shows that the model building with the smallest defect dataset still takes longer than two days.[3] Hence, using semantic features for line-level defect prediction remains challenging.

5 Model-Agnostic-based Line-level Defect Prediction

Similar to prior work [16, 55, 59, 77, 80, 89, 90], the key goal of this work is to help a software development team develop an effective SQA resource management by priortizing the limited SQA effort on the most defect-prone areas of source code. Rather than attempting to build a line-level defect model, we hypothesize that a prediction of a file-level defect model can be further explained to identify defect-prone lines. Recently, model-agnostic techniques have been proposed to provide a local explanation for a prediction of any machine learning algorithms. The fundamental concept of the local explanation is to provide information why the model makes such a prediction. Unlike the traditional model interpretation techniques (TMI) like variable importance for random forest [9] or the coefficients analysis for logistic regression models [27], the model-agnostic techniques can identify important features for a given file by estimating the contribution of each token feature to a prediction of the model. Figure 3 illustrates the difference of important features that are identified by the TMI and model-agnostic techniques. The key difference is that the TMI techniques will generate only one set of important features based on the models that are trained on a given training dataset, while the model-agnostic technique will generate a set of important features for each testing file.

To leverage the model-agnostic techniques to identify defect-prone lines, we propose a Model Agnostic-based Line-level Defect Prediction framework (called LINE-DP).

3. The detail is provided in Appendix (Section 10.1).

Algorithm 1: LIME's algorithm [69]

```
Input  : $f$ is a prediction model,
         $x$ is a test instance,
         $n$ is a number of randomly generated
         instances, and
         $k$ is a length of explanation
Output: $E$ is a set of contributions of features on the
         prediction of the instance $x$.
1 $D = \varnothing$
2 for $i$ in $\{1, ..., n\}$ do
3     $d_i = \text{GenInstAroundNeighbourhood}(x)$
4     $y'_i = \text{Predict}(f, d_i)$
5     $D = D \cup \langle d_i, y'_i \rangle$
6 end
7 $l = \text{K-Lasso}(D, k)$
8 $E = \text{get\_coefficients}(l)$
9 return $E$
```

To do so, we first use source code tokens of a file as features (i.e., token features) to build a file-level defect model. Then, we generate a prediction for each testing file using the file-level defect model. Then, we use a state-of-the-art model-agnostic technique, i.e., Local Interpretable Model-Agnostic Explanations (LIME) [69] to generate an explanation for a prediction of the file-level defect models. More specifically, given a testing file, LIME will identify important token features that influence the file-level defect model to predict that the testing file will be defective. Finally, we rank the defect-prone lines based on LIME scores instead of the defect-proneness of files. Our intuition is that code tokens that frequently appeared in defective files in the past may also appear in the lines that will be fixed after release.

Figure 4 presents an overview of our framework. Below, we provide the background of the Local Interpretable Model-agnostic Explanations (LIME) algorithm and describe the details of our proposed framework.

5.1 Local Interpretable Model-agnostic Explanations (LIME)

LIME is a model-agnostic technique that aims to mimic the behavior of the predictions of the defect model by explaining the individual predictions [69]. Given a file-level defect model $f()$ and a test instance x (i.e., a testing file), LIME will perform three main steps: (1) generating the neighbor instances of x; (2) labelling the neighbors using $f()$; (3) extracting local explanations from the generated neighbors. Algorithm 1 formally describes the LIME algorithm. We briefly describe each step as follows:

1) **Generate neighbor instances of a test instance x.** LIME randomly generates n synthetic instances surrounding the test instance x using a random perturbation method with an exponential kernel function on cosine distance (*cf.* Line 3).
2) **Generate labels of the neighbors using a file-level defect model f.** LIME uses the file-level defect model f to generate the predictions of the neighbor instances (*cf.* Line 4).
3) **Generates local explanations from the generated neighbors.** LIME builds a local sparse linear regres-

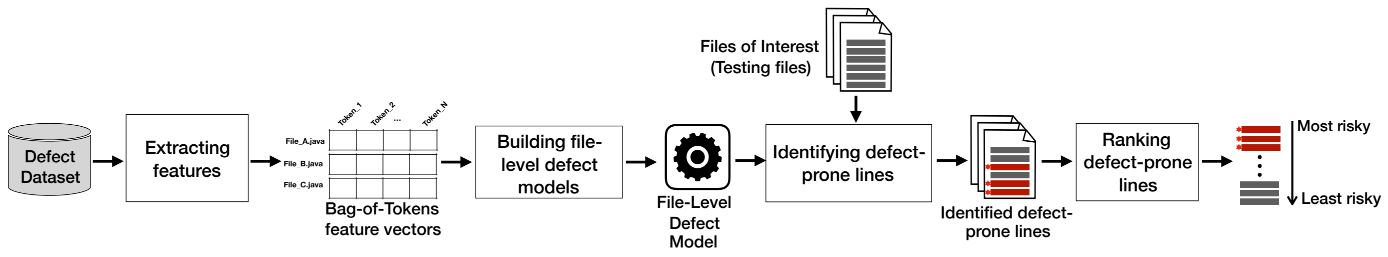


Fig. 4: An overview of our approach of localizing defective lines.

sion model (K-Lasso) using the randomly generated instances and their generated predictions from the file-level defect model f (*cf.* Line 7). The coefficients of the K-Lasso model (l) indicate the importance score of each feature on the prediction of a test instance x according to the prediction model l (*cf.* Line 8).

The importance score (e) of each feature in E ranges from -1 to 1. A positive LIME score of a feature ($0 < e \leq 1$) indicates that the feature has a positive impact on the estimated probability of the test instance x. On the other hand, a negative LIME score of a feature ($-1 \leq e < 0$) indicates that the feature has a negative impact on the estimated probability.

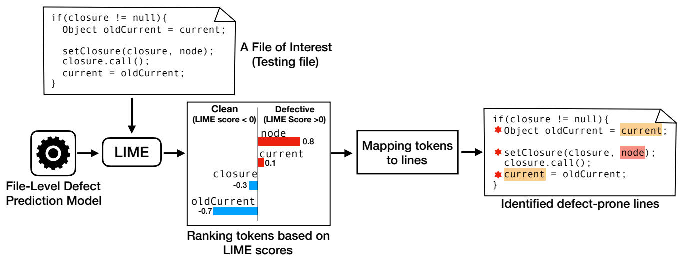
5.2 Our Line-DP Framework

Figure 4 presents an overview of our Model Agnostic-based Line-level Defect Prioritization (Line-DP) framework. Given a file-level defect dataset (i.e., a set of source code files and a label of defective or clean), we first extract bag-of-token features for each file. Then, we train traditional machine learning techniques (e.g., logistic regression, random forest) using the extracted features to build a file-level defect model. We then use the file-level defect model to estimate the probability that a testing file will be defective. For each file that is predicted as defective (i.e., defect-prone files), we use LIME to identify and rank defect-prone lines based on the LIME scores. We describe each step below.

(Step 1) Extracting Features

In this work, we use code tokens as features to represent source code files. This will allow us to use LIME to identify the tokens that lead the file-level defect models to predict that a given file will be defective. To do so, for each source code file in defect datasets, we first apply a set of regular expressions to remove non-alphanumeric characters such as semi-colon (;), equal sign (=). As suggested by Rahman and Rigby [66], removing these non-alphanumeric characters will ensure that the analyzed code tokens will not be artificially repetitive. Then, we extract code tokens in the files using the `Countvectorize` function of the Scikit-Learn library. We neither perform lowercase, stemming, nor lemmatization (i.e., a technique to reduce inflectional forms) on our extracted tokens, since the programming language of our studied systems (i.e., Java) is case-sensitive. Thus, meaningful tokens may be discarded when applying stemming and lemmatization.

After we extract tokens in the source code files, we use a bag of tokens (BoT) as a feature vector to represent a source



```
if(closure != null){
  Object oldCurrent = current;

  setClosure(closure, node);
  closure.call();
  current = oldCurrent;
}
```

```
if(closure != null){
*Object oldCurrent = current;

*setClosure(closure, node);
 closure.call();
*current = oldCurrent;
}
```

Fig. 5: An illustrative example of our approach for identifying defect-prone lines.

code file. A bag of tokens is a vector of frequencies that code tokens appear in the file. To reduce the sparsity of the vectors, we remove tokens that appear only once.

(Step 2) Building File-Level Defect Models

We build a file-level defect model using the feature vectors extracted in Step 1. Prior work suggests that the performance of defect models may vary when using different classification techniques [22]. Hence, in this work, we consider five well-known classification techniques [22, 77, 80], i.e., Random Forest (RF), Logistic Regression (LR), Support Vector Maching (SVM), k-Nearest Neighbours (kNN), and Neural Networks (NN). We use the implementation of Python Scikit-Learn package to build our file-level defect models using default parameter settings. Based on the predictive performance at the file level, we find that the file-level defect models that use *Logistic Regression* can identify actual defective files relatively better than other four classification techniques, achieving a median MCC value of 0.35 (within-release) and 0.18 (cross-release).[4] We consider that the accuracy of our file-level defect models is sufficient since prior study reported that a file-level prediction model typically has an MCC value of 0.3 [71]. Hence, in this paper, our Line-DP is based on a file-level defect model that uses *Logistic Regression*.

(Step 3) Identifying Defect-Prone Lines

For the defect-prone files predicted by our file-level defect models (a probability > 0.5), we further identify defect-prone lines using LIME [69]. Figure 5 provides an illustrative example of our approach. Given a defect-prone file, we use LIME to compute LIME scores, i.e., an importance score of features (code tokens). We identify the tokens that have a positive LIME scores as **risky tokens**. For example,

4. We provide complete evaluation results of the file-level defect models in Appendix (Section 10.2).

in Figure 5, `node` and `current` have a LIME score of 0.8 and 0.1, respectively. Hence, these two tokens are identified as risky tokens. Then, we define a **defect-prone line** as the line that contains at least one of the risky tokens. For example, in Figure 5, the lines that are marked by star polygon contain `node` and `current`. Therefore, these three lines are identified as defect-prone lines.

Considering all positive LIME scores may increase false positive identification. Therefore, in this paper, we use top-20 risky tokens ranked based on LIME scores when identifying defect-prone lines. The number of top risky tokens (k) is selected based on a sensitivity analysis where $10 \leq k \leq 200$.[5]

(Step 4) Ranking Defect-Prone Lines

Once we identify defect-prone lines in all predicted defective files, we now rank defect-prone lines based on the number of the top-20 risky tokens that appear in the defect-prone lines. The intuition behind is that the more the risky tokens that a line contains, the more likely the line will be defective. For example, given two defect-prone lines $l_1 = \{A, B, C, D\}$ and $l_2 = \{C, D, E, F, G\}$, where $A - G$ denote code tokens and tokens A, B and E are the top-20 risky tokens. Then, line l_1 should be given a higher priority than line l_2 as l_1 contains two risky tokens and l_2 contains only one risky token.

6 Experimental Setup

In this section, we describe our studied software systems, an approach to extract defective lines, baseline approaches, evaluation measures, and validation settings.

6.1 Studied Software Systems

In this work, we use a corpus of publicly-available defect datasets provided by Yatish *et al.* [91] where the ground-truths are labelled based on the affected releases. The datasets consist of 32 releases that span 9 open-source software systems from the Apache open source software projects. Table 1 shows a statistical summary of the studied datasets. The number of source code files in the datasets ranges from 731 to 8,846, which have 74,349 - 567,804 lines of code, and 58,659 - 621,238 code tokens.

6.2 Extracting Defective Lines

We now describe an approach for extracting defective lines. Similar to prior work [64, 67], we identify that *defective lines are those lines that were changed by bug-fixing commits*. Figure 6 provides an illustrative example of our approach, which we describe in details below.

Identifying bug-fixing commits: We first retrieve bug reports (i.e., the issue reports that are classified as “Bug” and that affect the studied releases) from the JIRA issue tracking systems of the studied systems. We then use the ID of these bug reports to identify bug-fixing commits in the Git Version Control Systems (VCSs). We use regular expressions to search for the commits that have the bug report IDs

5. We provide the results of our sensitivity analysis in Appendix (Section 10.3)

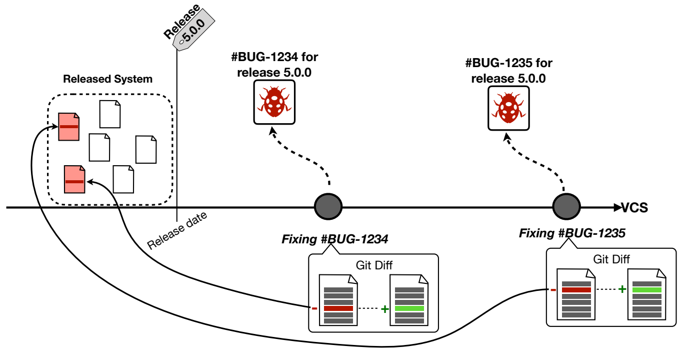


Fig. 6: An illustrative example of our approach for extracting defective lines.

in the commit messages. Those commits that have the ID of a bug report are identified as bug-fixing commits. This technique allows us to generate a defect dataset with fewer false positives than using a defect-related keyword search (like “bug”, “defect”) which lead to a better performance of defect prediction models [63, 91].

Identifying defective lines: To identify defective lines, we first examine the diff (a.k.a. code changes) made by bug-fixing commits. We use the implementation of PyDriller package to extract such information from Git repositories [72]. Similar to prior work [64, 67], the lines that were modified by bug-fixing commits are identified as ***defective lines***. We only consider the modified lines that appear in the source files at the studied release. Other lines that are not impacted by the bug-fixing commits are identified as ***clean lines***. Similar to Yatish *et al.* [91], we also identify the files that are impacted by the bug-fixing commits as *defective files*, otherwise clean.

6.3 Baseline Approaches

In this work, we compare our LINE-DP against six approaches, i.e., random guessing, two static analysis tools, an NLP-based approach, and two traditional model interpretation (TMI) based approaches. We describe the baseline approaches below.

Random Guessing. Random guessing has been used as a baseline in prior work [64, 67]. To randomly select defect-prone lines, we first use the file-level defect model to identify defect-prone files. Then, instead of using LIME to compute a LIME score, we assign a random score ranging from -1 to 1 to each token in those defect-prone files. The tokens with a random score greater than zero are identified as risky tokens. Finally, the line is identified as *defect-prone lines* if it contains at least one of the top-20 risky tokens based on the random scores. We then rank defect-prone lines randomly similar to prior work [67].

Static Analysis. Prior work shows that a static analysis tool can be used to identify defect-prone lines [25, 42, 47, 64, 66, 67]. Habib and Pradel [25] found that static bug detectors are certainly worthwhile to detect real-world bugs. Hence, we use two static analysis tools, i.e., PMD [12] and ErrorProne [2] as our baseline approaches.

PMD: We use PMD [12] which is often used in previous research [42, 47, 64, 66, 67]. We did not use FINDBUGS since prior studies [66, 67] show that the performance of PMD

TABLE 1: An overview of the studied systems.

System	Description	#Files	#LOC	#Code Tokens	%Defective Files	Studied Releases
ActiveMQ	Messaging and Integration Patterns	1,884-3,420	142k-299k	141k-293k	2%-7%	5.0.0, 5.1.0, 5.2.0, 5.3.0, 5.8.0
Camel	Enterprise Integration Framework	1,515-8,846	75k-485k	94k-621k	2%-8%	1.4.0, 2.9.0, 2.10.0, 2.11.0
Derby	Relational Database	1,963-2,705	412k-533k	251k-329k	6%-28%	10.2.1.6, 10.3.1.4, 10.5.1.1
Groovy	Java-syntax-compatible OOP	757-884	74k-93k	58k-68k	2%-4%	1.5.7, 1.6.0.Beta_1, 1.6.0.Beta_2
HBase	Distributed Scalable Data Store	1,059-1,834	246k-537k	149k-257k	7%-11%	0.94.0, 0.95.0, 0.95.2
Hive	Data Warehouse System for Hadoop	1,416-2,662	290k-567k	147k-301k	6%-19%	0.9.0, 0.10.0, 0.12.0
JRuby	Ruby Programming Lang for JVM	731-1,614	106k-240k	72k-165k	2%-13%	1.1, 1.4, 1.5, 1.7
Lucene	Text Search Engine Library	805-2,806	101k-342k	76k-282k	2%-8%	2.3.0, 2.9.0, 3.0.0, 3.1.0
Wicket	Web Application Framework	1,672-2,578	106k-165k	93k-147k	2%-16%	1.3.0.beta1, 1.3.0.beta2, 1.5.3

and FINDBUGS are comparable. PMD is a static analysis tool that identifies common errors based on a set of predefined rules with proven properties. Given a source code file, PMD checks if source code violates the rules and reports warnings which indicate the violated rules, priority, and the corresponding lines in that file. Similar to prior work [66, 67], we identify the lines reported in the warnings as *defect-prone lines*. We rank the defect-prone lines based on the priority of the warnings where a priority of 1 indicates the highest priority and 4 indicates the lowest priority.

ErrorProne (EP): Recently, major companies, e.g., Google, use ErrorProne to identify defect-prone lines [2]. ErrorProne is a static analysis tool that builds on top of a primary Java compiler (javac) to check errors in source code based on a set of error-prone rules. ErrorProne checks if a given source code file is matched error-prone rules using all type attribution and symbol information extracted by the compiler. The report of ErrorProne includes the matched error-prone rules, suggestion messages, and the corresponding lines in the file. In this experiment, we identify the corresponding lines reported by ErrorProne as *defect-prone lines*. Since ErrorProne does not provide priority of the reported errors like PMD, we rank the defect-prone lines based on the line number in the file. This mimics a top-down reading approach, i.e., developers sequentially read source code from the first to last lines of the files.

NLP-based Approach. Ray *et al.* [67] have shown that entropy estimated by n-gram models can be used to rank defect-prone files and lines. Hence, we compute entropy for each code token in source code files based on the probability estimated by n-gram models. In this work, we use an implementation of Hellendoorn and Devanbu [31] to build cache-based language models, i.e., an enhanced n-gram model that is suitable for source code. We use a standard n-gram order of 6 with the Jelinek-Mercer smoothing function as prior work demonstrates that this configuration works well for source code [31]. Once we compute entropy for all code tokens, we compute average entropy for each line. The lines that have average entropy greater than a threshold are identified as *defect-prone lines*. In this experiment, the entropy threshold is 0.7 and 0.6 for the within-release and cross-release validation settings, respectively.[6] Finally, we rank defect-prone lines based on their average entropy.

Traditional Model Interpretation (TMI)-based Approach. TMI techniques can be used to identify the important features in the defect models [8]. However, the TMI techniques will provide only one set of important features for all files of interest, e.g., testing files (see Figure 3). Nevertheless, ones might use TMI techniques to identify defect-prone lines like our LINE-DP approach. Hence, we build TMI-based approaches using two classifcation techniques: Logistic Regression (TMI-LR) and Random Forest (TMI-RF) as our baseline approaches.

TMI-LR: To identify defect-prone lines using the TMI-based approach with Logistic Regression (LR), we examine standardized coefficients in our logistic regression models. Unlike the simple coefficients, the standardized coefficients can be used to indicate the contribution that a feature made to the models regardless the unit of measurement, which allows us to compare the contribution among the features [52]. The larger the positive coefficient that the feature has, the larger the contribution that the feature made to the model. To examine standardized coefficients, we use the `StandardScalar` function of the Scikit-Learn Python library to standardize the token features. Then, we use the coefficient values of the standardized token features in the logistic regression models to identify risky tokens. More specifically, the tokens with a positive coefficient are identified as risky tokens. Then, for the testing files, we identify the lines as *defect-prone lines* when those lines contain at least one of the top-20 risky tokens based on the coefficient values. Finally, we rank the defect-prone lines based on the number of the top-20 risky tokens that appear in the defect-prone lines similar to our LINE-DP approach.

TMI-RF: To identify defect-prone lines using the TMI-based approach with Random Forest (RF), we examine feature importance in the model, i.e., the contribution of features to the decision making in the model. The larger the contribution that a feature (i.e., a token) made to the model, the more important the feature is. To do so, we use the `feature_importances_` function of the Scikit-Learn Python library which is the impurity-based feature importance measurement. We identify defect-prone lines based on the feature importance values of the tokens. In this experiment, the top-20 important token features are identified as risky tokens. Then, for the testing files, the lines are identified as *defect-prone lines* if they contain at least one of the top-20 important token features. Similar to our LINE-DP approach, we rank the defect-prone lines based on the number of the top-20 important token features that appear in the defect-prone lines.

6.4 Evaluation Measures

To evaluate the approaches, we use five performance measures preferred by practitioners [87], i.e., recall, false alarm rate, a combination of recall and false alarm rate, initial

6. The sensitivity analysis and its results are described in Appendix (Section 10.4).

false alarm, and Top k%LOC Recall.[7] In addition, we use Matthews Correlation Coefficients (MCC) to evaluate the overall predictive accuracy which is suitable for the unbalanced data like our line-level defect datasets [8, 71]. Below, we describe each of our performance measures.

Recall. Recall measures the proportion between the number of lines that are correctly identified as defective and the number of actual defective lines. More specifically, we compute recall using a calculation of $\frac{TP}{(TP+FN)}$, where TP is the number of actual defective lines that are predicted as defective and FN is the number of actual defective lines that are predicted as clean. A high recall value indicates that the approach can identify more defective lines.

False alarm rate (FAR). FAR measures a proportion between the number of clean lines that are identified as defective and the number of actual clean lines. More specifically, we measure FAR using a calculation of $\frac{FP}{(FP+TN)}$, where FP is the number of actual clean lines that are predicted as defective and TN is the number of actual clean lines that are predicted as clean. The lower the FAR value is, the fewer the clean lines that are identified as defective. In other words, a low FAR value indicates that developers spend less effort when inspecting defect-prone lines identified by the an approach.

A combination of recall and FAR. In this work, we use *Distance-to-heaven (d2h)* of Agrawal and Menzies [4] to combine the recall and FAR values. D2h is the root mean square of the recall and false alarm values (i.e., $\sqrt{\frac{(1-Recall)^2+(0-FAR)^2}{2}}$) [3, 4]. A d2h value of 0 indicates that an approach achieves a perfect identification, i.e., an approach can identify all defective lines (Recall $= 1$) without any false positives (FAR $= 0$). A high d2h value indicates that the performance of an approach is far from perfect, e.g., achieving a high recall value but also have high a FAR value and vice versa.

Top k%LOC Recall. Top k%LOC recall measures how many actual defective lines found given a fixed amount of effort, i.e., the top k% of lines ranked by their defect-proneness [34]. A high value of top k%LOC recall indicates that an approach can rank many actual defective lines at the top and many actual defective lines can be found given a fixed amount of effort. On the other hand, a low value of top k% LOC recall indicates many clean lines are in the top k% LOC and developers need to spend more effort to identify defective lines. Similar to prior work [43, 53, 64, 67], we use 20% of LOC as a fixed cutoff for an effort.

Initial False Alarm (IFA). IFA measures the number of clean lines on which developers spend SQA effort until the first defective line is found when lines are ranked by their defect-proneness [34]. A low IFA value indicates that few clean lines are ranked at the top, while a high IFA value indicates that developers will spend unnecessary effort on clean lines. The intuition behinds this measure is that developers may stop inspecting if they could not get promising results (i.e., find defective lines) within the first few inspected lines [60].

7. Note that we have confirmed with one of the authors of the survey study [87] that top k%LOC Recall is one of the top-5 measures, not top k%LOC Precision as reported in the paper.

Matthews Correlation Coefficients (MCC). MCC measures a correlation coefficients between actual and predicted outcomes using the following calculation:

$$\frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \qquad (1)$$

An MCC value ranges from -1 to +1, where an MCC value of 1 indicates a perfect prediction, and -1 indicates total disagreement between the prediction

6.5 Validation Settings

In this paper, we perform both within-release and cross-release validation settings. Below, we describe each of our validation settings.

Within-release setting. To perform within-release validation, we use the stratified 10×10-fold cross validation technique for each release of the studied systems. To do so, we first randomly split the dataset of each studied release into 10 equal-size subsets while maintaining the defective ratio using the `StratifiedShuffleSplit` function of the Scikit-Learn Python library. The stratified k-fold cross validation tends to produce less bias for estimating the predictive accuracy of a classification model than the traditional 10-fold cross validation [79]. For each fold of the ten folds, we use it as a testing dataset and use the remaining nine folds to train the models (e.g., the file-level defect models, n-gram models). To ensure that the results are robust, we repeat this 10-fold cross-validation process 10 times, which will generate 100 performance values. Finally, we compute an average of those 100 values to estimate the performance value of the approach.

Cross-release setting. To mimic a practical usage scenario of defect prediction models to prioritize SQA effort, we use the cross-release setting by considering a time factor (i.e., the release date) when evaluating an approach. The goal of this validation is to evaluate whether an approach can use the dataset of the past release ($k-1$) to identify defect-prone lines in the current release (k) or not. More specifically, we use the dataset of release $k-1$ to train the models (e.g., the file-level defect models, n-gram models). Then, we use the dataset of the release k as a testing dataset to evaluate the approaches. For example, we build the models using the dataset of ActiveMQ 5.0.0 and use the dataset of ActiveMQ 5.1.0 to evaluate the models. We perform this evaluation for every pair of the consecutive releases of a studied system. For 32 studied releases shown in Table 1, we have 23 pairs of consecutive releases for our cross-release validation.

6.6 Statistical Analysis

We now describe our approaches to analyze the performance of our LINE-DP against each baseline approach.

Performance Gain. To determine whether our LINE-DP is better than the baseline approaches, we compute the percentage of the performance difference between our LINE-DP and each of the baseline approaches using the following calculation:

$$\%\text{PerformanceDiff} = \frac{\sum(\text{Perf}_{\text{LINE-DP}} - \text{Perf}_{baseline})}{\sum \text{Perf}_{baseline}} \qquad (2)$$

A positive value of the percentage difference indicates that the performance of our LINE-DP is greater than the baseline approaches, while a negative value indicates that the performance our LINE-DP is lower than the baseline approaches.

Statistical Test. We use the one-sided Wilcoxon-signed rank test to confirm the statistical difference. More specifically, we compare the performance of our LINE-DP against each baseline approach (i.e., LINE-DP *vs* Random, LINE-DP *vs* PMD). We use a Wilcoxon signed-rank test because it is a non-parametric statistical test which performs a pairwise comparison between two distributions. This allows us to compare the performance of our LINE-DP against the baseline approach based on the same release for the within-release setting (or the same train-test pair for the cross-release setting). We use the `wilcoxsign_test` function of the R coin library. We also measure the effect size (r) i.e., the magnitude of the difference between two distributions using a calculation of $r = \frac{Z}{\sqrt{n}}$ where Z is a statistic Z-score from the Wilcoxon signed-rank test and n is the total number of samples [83]. The effect size $r > 0.5$ is considered as large, $0.3 < r \leq 0.5$ is medium, and $0.1 < r \leq 0.3$ is small, otherwise negligible [18]. We did not use the commonly-used Cohen's D [11] and Cliff's δ [51] to measure the effect size because both methods are not based on the assumption that the data is pairwise.

7 EVALUATION RESULTS

In this section, we present the approach and results for each of our research questions.

(RQ1) How effective is our LINE-DP to identify defective lines?

Motivation. Our preliminary analysis shows that only 1%-3% of lines in a source code file are defective (see Section 3), suggesting that developers could waste a relatively large amount of their effort on inspecting clean lines. Prior work also argues that it may not be practical when predicting defects at the coarse-grained level even if the defect models achieve high accuracy than fine-grained granularity level of predictions [28]. Thus, a defect prediction model that identifies defect-prone lines (i.e., lines that are likely to be defective after release) would be beneficial to an SQA team to focus on the defect-prone lines. Hence, in this RQ, we set out to investigate how well our LINE-DP can identify defective lines.

Approach. To answer our RQ1, we use our LINE-DP and six baseline approaches (see Section 6.3) to predict defective lines in the given testing files. We evaluate our LINE-DP and the baseline approaches using the within-release and cross-release validation settings. To measure the predictive accuracy, We use Recall, False Alarm Rate (FAR), Distance-2-heaven (d2h), and the Matthews Correlation Coefficients (MCC) (see Section 6.4). We did not specifically evaluate the precision of the approaches because the main goal of this work is *not* to identify exact defective lines, but instead to help developers reduce the SQA effort by scoping down the lines that require SQA. Moreover, focusing on maximizing precision values would leave many defective lines unattended from SQA activities.

Finally, we perform a statistical analysis to compare the performance between our LINE-DP and the baseline approaches (see Section 6.6). More specifically, we use the one-sided Wilcoxon signed-rank test to confirm whether the recall and MCC values of our LINE-DP are significantly higher than the baseline approaches; and whether the FAR and d2h values of our LINE-DP are significantly lower than the baseline approaches.

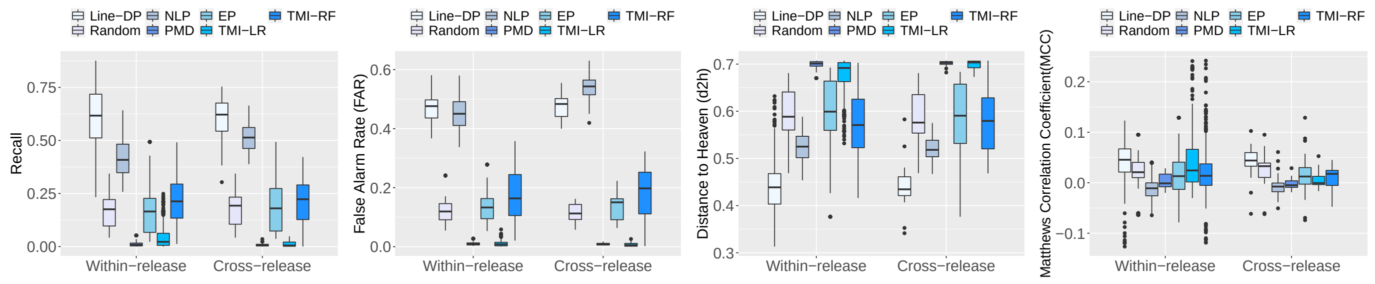
Results. Figure 7a shows that at the median, our LINE-DP achieves a recall of 0.61 and 0.62 based on the within-release and cross-release settings, respectively. This result indicates that 61% and 62% of actual defective lines in a studied release can be identified by our LINE-DP. Figure 7b also shows that our LINE-DP has a FAR of 0.47 (within-release) and 0.48 (cross-release) at the median values. This result suggests that when comparing with the traditional approach of predicting defects at the file level, our LINE-DP could potentially help developers reduce SQA effort that will be spent on 52% of clean lines, while 62% of defective lines will be examined.

Figure 7a shows that our LINE-DP achieves the most promising results, compared to the six baseline approaches for both within-release and cross-release settings. Moreover, Table 2 shows that the recall values of our LINE-DP are 44%-4,871% (within-release) and 18%-6,691% (cross-release) larger than the recall values of the baseline approaches. The one-sided Wilcoxon signed-rank tests also confirm the significance (p-value < 0.01) with a medium to large effect size.

On the other hand, Figure 7b shows that our LINE-DP has a FAR value larger than the baseline approaches. Table 2 shows that only the NLP-based approach that has a FAR value 15% larger than our LINE-DP for the cross-release setting. The lower FAR values of the baseline approaches because of the lower number of lines that are predicted as defective. Indeed, at the median, 0 - 77 of lines in a file are predicted as defective by the baseline approaches, while 90 - 92 of the lines are predicted as defective by our LINE-DP. Intuitively, the fewer the predicted lines are, the less likely that the technique will give a false prediction. Yet, many defective lines are still missed by the baseline approaches according to the recall values which are significantly lower than our LINE-DP. Hence, the performance measures (e.g., distance-to-heaven) that concern both aspects should be used to compare the studied approaches.

Figure 7c shows that, at the median, our LINE-DP achieves a median d2h value of 0.44 (within-release) and 0.43 (cross-release), while the baseline approaches achieve a median d2h value of 0.52 to 0.70. Table 2 shows that our LINE-DP have the d2h values 16%-37% (within-release) and 15%-37% lower than the baseline approaches. The one-sided Wilcoxon-signed rank tests also confirm the statistical significance (p-value < 0.001) with a large effect size. These results indicate that when considering both the ability of identifying defective lines (i.e., recall) and the additional costs (i.e., FAR), our LINE-DP outperforms the baseline approaches.

Table 2 also shows that our LINE-DP also achieves MCC significantly better than the baseline approaches. The one-sided Wilcoxon-signed rank tests also confirm the statistical significance (p-value < 0.001) with a medium to large ef-



(a) Recall (b) False Alarm (c) Distance-to-Heaven (d) Matthews Correlation Coefficients

Fig. 7: Distributions of Recall, FAR, D2H, and MCC values of our LINE-DP and the baseline approaches.

TABLE 2: A comparative summary of the predictive accuracy between our LINE-DP and the baseline approaches. The bold text indicates that our LINE-DP is better than the baseline approaches.

Within-release validation

LINE-DP vs	Recall ↗		FAR ↘		d2h ↘		MCC ↗	
Baseline	%Diff	Eff. Size (r)	%Diff	Eff. Size (r)	%Diff	Eff. Size (r)	%Diff	Eff. Size (r)
Random	**260%**	L***	298%	∘	**-26%**	L***	**91%**	M**
PMD	**4,871%**	L***	4,712%	∘	**-37%**	L***	**1,411%**	M***
EP	**240%**	L***	250%	∘	**-25%**	L***	**264%**	M***
NLP	**44%**	M***	4%	∘	**-16%**	L***	**484%**	L***
TMI-LR	**1,225%**	L***	4,112%	∘	**-35%**	L***	-9%	∘
TMI-RF	**180%**	L***	173%	∘	**-23%**	L***	**80%**	M***

Cross-release validation

LINE-DP vs	Recall ↗		FAR ↘		d2h ↘		MCC ↗	
Baseline	%Diff	Eff. Size (r)	%Diff	Eff. Size (r)	%Diff	Eff. Size (r)	%Diff	Eff. Size (r)
Random	**243%**	L***	303%	∘	**-25%**	L***	**72%**	M**
PMD	**6,691%**	L***	5,159%	∘	**-37%**	L***	**2,754%**	L***
EP	**226%**	L***	254%	∘	**-25%**	L***	**149%**	M**
NLP	**18%**	M**	**-12%**	L***	**-15%**	L***	**914%**	L***
TMI-LR	**5,079%**	L***	5,966%	∘	**-37%**	L***	**639%**	L***
TMI-RF	**190%**	L***	163%	∘	**-24%**	L***	**308%**	M***

Effect Size: Large (L) $r > 0.5$, Medium (M) $0.3 < r \leq 0.5$, Small (S) $0.1 < r \leq 0.3$, Negligible (N) $r < 0.1$
Statistical Significance: $^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$, $\circ p \geq 0.05$

fect size. Figure 7d shows that at the median, our LINE-DP achieves an MCC value of 0.05 (within-release) and 0.04 (cross-release), while the baseline approaches achieve an MCC value of -0.01 - 0.02 (within-release) and -0.01 - 0.03 (cross-release). These results suggest that our LINE-DP achieves a better predictive accuracy than the baseline approaches.

Nevertheless, our LINE-DP still achieves a relatively low MCC value. This is because our LINE-DP still produces high false positives, i.e., many clean lines are predicted as defective. Given a very small proportion of defective lines (i.e., only 1% - 3%) in a file, it is challenging to identify exact defective lines without any false positives. Moreover, the main goal of this work is *not* to identify exact defective lines, but instead to help developers reduce the SQA effort by scoping down the lines that require SQA. Then, focusing on minimizing false positives may leave many defective lines unattended from SQA activities. Considering the d2h value, we believe that our LINE-DP is still of value to practitioners (i.e., achieving a relatively high recall given the false positives that the approach produced).

(RQ2) How well can our LINE-DP rank defective lines?

Motivation. One of the key benefits of defect prediction is to help developers perform a cost-effective SQA activity by priortizing defect-prone files in order to uncover maximal defects with minimal effort [28, 43, 53, 61]. In other words, an effective prioritization should rank defective lines to the top in order to help developers find more defects given the limited amount of effort. Thus, we set out to investigate the ranking performance of LINE-DP. More specifically, we evaluate how many defective lines can be identified given the fixed amount of effort (i.e., Top k%LOC Recall) and how many clean lines (i.e., false positives) will be unnecessarily examined before the first defective line is found (i.e., Initial False Alarm). The intuition behinds is that developers may stop following a prediction if they could not get promising results (i.e., find defective lines) given a specific amount of effort or within the first few inspected lines [60].

Approach. To answer our RQ2, we rank the defect-prone lines based on our approach (see Section 5.2) and the baseline approaches (see Section 6.3). To evaluate the ranking performance, we use top k%LOC recall and Initial False Alarm (IFA) (see Section 6.4). Top k%LOC recall measures the proportion of defective lines that can be identified given a fixed amount of k% of lines. Similar to prior work [43, 53, 64, 67], we use 20% of LOC as a fixed cutoff for an effort. IFA counts how many clean lines are inspected until the first defective line is found when inspecting the lines ranked by the approaches. We evaluate the ranking

TABLE 3: A comparative summary of the ranking performance between our LINE-DP and the baseline approaches. The bold text indicates that our LINE-DP is better than the baseline approaches.

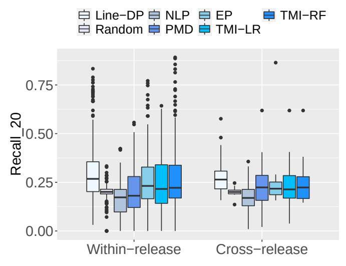
Within-release validation

LINE-DP vs.	Recall@Top20% ↗		IFA ↘	
Baseline	%Diff	Eff. Size (r)	%Diff	Eff. Size (r)
Random	**53%**	M***	**-23%**	○
PMD	**46%**	M***	**-55%**	M***
EP	**18%**	S*	**-50%**	M***
NLP	**91%**	M***	**-94%**	L***
TMI-LR	**22%**	M**	**-43%**	○
TMI-RF	**11%**	S*	**-70%**	M***

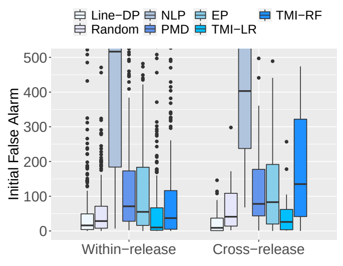
Cross-release validation

LINE-DP vs.	Recall@Top20% ↗		IFA ↘	
Baseline	%Diff	Eff. Size (r)	%Diff	Eff. Size (r)
Random	**42%**	L***	**-51%**	M*
PMD	**22%**	M*	**-82%**	L***
EP	**17%**	M*	**-78%**	L***
NLP	**68%**	M***	**-99%**	L***
TMI-LR	**19%**	M*	**-29%**	○
TMI-RF	**17%**	M*	**-89%**	M***

Effect Size: Large (L) $r > 0.5$, Medium (M) $0.3 < r \leq 0.5$, Small (S) $0.1 < r \leq 0.3$, Negligible (N) $r < 0.1$
Statistical Significance: $^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$, $\circ p \geq 0.05$



(a) Recall@Top20%LOC



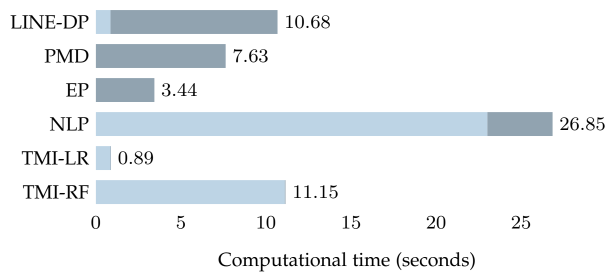
(b) Initial False Alarm

Fig. 8: Distributions of Initial False Alarm values and a proportion of defective lines found at the fixed effort (i.e., 20% of lines) of our LINE-DP and the baseline approaches.

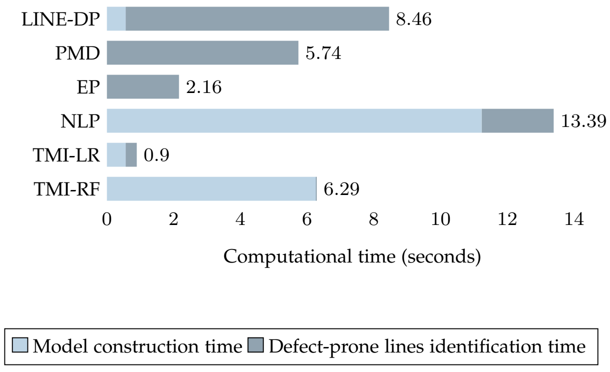
performance based on both within-release and cross-release settings. Similar to RQ1, we use the one-sided Wilcoxon signed-rank test to confirm whether the top 20%LOC recall values of our LINE-DP are significantly higher than the baseline approaches; and whether the IFA values of LINE-DP are significantly lower than the baseline approaches.

Results. Figure 8a shows that, at the median, our LINE-DP achieves a recall of 0.27 (within-release) and 0.26 (cross-release) if top 20% of the total lines are examined. On the other hands, the baseline approaches achieve a lower top 20%LOC recall with a median of 0.17 - 0.22. Table 3 shows that the top 20%LOC recall values of our LINE-DP are 22% - 91% (within-release) and 19% - 68% (cross-release) larger than those of the baseline approaches. The one-sided Wilcoxon-signed rank tests also confirm the statistical significance (p-value < 0.05) with a medium to large effect size. These results suggest that our LINE-DP can rank defective lines better than the baseline approaches.

Figure 8b shows that at the median, our LINE-DP has a median IFA value of 16 (within-release) and 9 (cross-release), while the baseline approaches have a median IFA value of 10 - 517 (within-release) and 26 - 403 (cross-release).



(a) With-release setting



(b) Cross-release setting

Fig. 9: The average computation time (seconds) of our approach and baseline approaches.

Table 3 also shows that the IFA values of our LINE-DP are 23%-94% (within-release) and 29%-99% smaller than the baseline approaches. The one-sided Wilcoxon-signed rank tests confirm the statistical significance (p-value < 0.05) with a medium to large effect size for our LINE-DP against Static Analysis and NLP-based approaches. These results suggest that when using our LINE-DP, fewer clean lines will be inspected to find the first defective line.

(RQ3) How much computation time is required to predict defective lines?

Motivation. Fisher *et al.* [19] raise a concern that the increased complexity of data analytics may incur additional computation cost of building defect prediction models. Yet, many practitioners [19] still prefer simple and fast solutions, but accurate. Thus, we set out to investigate the computational cost of identifying defective lines of our LINE-DP when compared to other approaches.

Approach. To address RQ3, we measure the computation time of the model construction and the identification of defect-prone lines for each approach. We measure the computation time for both within-release and cross-release settings. For the within-release setting, we measure an average computation time for 10×10-folds of all 32 studied releases. Similarly, we measure an average computation time for 23 pairs of the consecutive releases for the cross-release validation setting. The computational time is based on a standard computing machine with an Intel Core i9 2.3GHz and 16GB of RAM. Then, we report the statistical summary of the distribution of the computation time of each step for all studied defect datasets.

Results. Figure 9 presents the average computation time for the model construction and the identification of defect-prone lines for a given test file. The results show that the average computation time for our LINE-DP is 10.68 and 8.46 seconds for the within-release and cross-release settings, respectively. Figure 9 also shows that the NLP-based approach takes the computation times 251% ($\frac{26.85}{10.68}$) and 158% ($\frac{13.39}{8.46}$) longer than our LINE-DP, indicating that our LINE-DP makes a line-level prediction faster than the NLP-based approach. Although Figure 9 shows the static analysis tools (i.e., PMD and ErrorProne) and the TMI-based approaches take shorter time than our LINE-DP, the additional computational time of our LINE-DP should still be manageable when considering the predictive accuracy of defective lines.

(RQ4) What kind of defects can be identified by our LINE-DP?

Motivation. The key motivation of RQ4 is to qualitatively analyze the types of defects that our LINE-DP can identify. This analysis will provide a better understanding of the cases for which our LINE-DP can predict defective lines. Hence, we set out to examine the defective lines that our LINE-DP can and cannot identify.

Approach. We first identify a defective code block, i.e., consecutive lines that are impacted by bug-fixing commits. We examine a code block because it provides a clearer context and more information than a single defective line. Then, we examine the **hit defective blocks**, i.e., the code blocks of which all the defective lines can be identified by our LINE-DP; and the **missed defective blocks**, i.e., the code blocks of which none of the defective lines can be identified by our LINE-DP.

In this RQ, we conduct a manual categorization based on the cross-release setting because this setting mimics a more realistic scenario than the within-release setting. We obtain 6,213 hit blocks and 5,024 missed blocks from the dataset of 23 consecutive pairs across nine studied systems. Since the number of studied code blocks is too large to manually examine in its entirety, we randomly select a statistically representative sample of 362 hit blocks and 357 missed blocks for our analysis. These sample sizes should allow us to generalize the conclusion about the ratio of defect types to all studied code blocks with a confidence level of 95% and a confidence interval of 5%.[8]

We categorize a defect type for the sampled code blocks based on how the defect was fixed in the bug-fixing commits. We use a taxonomy of Chen *et al.* [10] which is summarized in Table 4. To ensure a consistent understanding of the taxonomy, the first four authors of this paper independently categorize defect types for the 30 hit and 30 missed defective blocks. Then, we calculate the inter-rater agreement between the categorization results of the four coders using Cohen's kappa. The kappa agreements are 0.86 and 0.81 for the hit and missed blocks, respectively, indicating that the agreement of our manual categorization is "almost perfect" [86]. Finally, the first author of this paper manually categorized the remaining blocks in the samples.

Results. Table 5 shows the proportion of defect types for the defective code blocks that can be identified by our LINE-DP (i.e., hit defective blocks) and that cannot be identified by our LINE-DP (i.e., missed defective blocks). The result shows that the majority types of defects for the hit defective blocks are argument change (32%) and condition change (18%), which account for 50% of the sampled data. Furthermore, Table 5 shows that 63% of the hit defective blocks can be categorized into the common defect types, while the remaining 37% of them are categorized as others. These results indicate that our LINE-DP can predict defect-prone lines that contain common defects.

On the other hand, Table 5 shows that the call changes and the target changes appear in the missed defective blocks more frequent than the hit defective blocks. Nevertheless, we observe that the defects in the missed defective blocks require a more complex bug fixing approach than the hit defective blocks. Table 5 also shows that 50% of the missed defective blocks cannot be identified in the common defect types. These results suggest that while our LINE-DP can identify the common defects (especially the argument changes and the condition changes), our LINE-DP may miss defects related to call changes, target changes, and other complex defects.

Furthermore, we observe that code tokens that frequently appear in defective files tend to be in the defective lines that will be fixed after the release. This is consistent with our intuition that code tokens that frequently appeared in defective files in the past may also appear in the lines that will be fixed after release. For example, "`runtime.getErr().print(msg);`" is a defective line where "`runtime`" is identified as a risky token by our LINE-DP. We observe that 90% of defective files ($\frac{30}{33}$) in the training dataset contain "`runtime`" token. Moreover, "`runtime`" is one of the 10 most frequent tokens in defec-

TABLE 4: A brief description of defect types.

Type	Description
Call change	Defective lines are fixed by modifying calls to method.
Chain change	The chaining methods in the defective lines are changed, added, or deleted.
Argument change	An argument of a method call in the defective lines are changed, added, or deleted.
Target change	A target that calls a method is changed in the defective lines.
Condition change	A condition statement in the defective lines is changed.
Java keyword change	A Java keyword in the defective lines is changed, added, and deleted.
Change from field to method call	A field assessing statement in the defective lines is changed to a method call statement.
Off-by-one	A classical off-by-one error in the defective lines.

TABLE 5: Defect types in our samples.

Defect type	Hit		Miss	
Argument change	116	(32%)	70	(20%)
Condition change	64	(18%)	13	(3%)
Call change	16	(4%)	46	(13%)
Java keyword change	16	(4%)	12	(3%)
Target change	13	(4%)	36	(10%)
Chain change	5	(1%)	2	(1%)
Others	132	(37%)	178	(50%)
Sum	**362**	**(100%)**	**357**	**(100%)**

8. https://www.surveysystem.com/sscalc.htm

tive files in the training dataset. Another example is that two out of three files that contain the "`filterConfig`" token are defective files in our training dataset. Then, our LINE-DP identifies "`filterConfig`" as a risky token for "`super.init(filterConfig)`" which was eventually fixed after the release. We provide examples of hit and missed defective blocks and their risky tokens for each defect type in Appendix (Section 10.5).

8 DISCUSSION

In this section, we discuss the limitation of our approach and possible threats to the validity of this study.

8.1 Limitation

The limitation of our LINE-DP is summarized as follow.

Our LINE-DP will produce many false positive predictions when common tokens become risky tokens. Our RQ2 shows that our LINE-DP has a false alarm rate (FAR) value larger than the baseline approaches. We observe that our LINE-DP will produce many false positive predictions when the common tokens (e.g., Java keywords or a generic identifier) are identified as risky tokens. This work opts to use a simple approach to select risky tokens, i.e., using top-k tokens based on a LIME score where k is selected based on the distance-to-heaven value. Future work should investigate an alternative approach to identify risky tokens, while lessening the interference of the common keywords.

Nevertheless, when considering all of the evaluation aspects other than false positives (i.e., recall, false alarm rate, d2h, the Top20%LOC Recall, Initial False Alarm), the empirical results show that LINE-DP significantly outperforms the state-of-the-art techniques that predict defective lines (i.e., NLP, ErrorProne, PMD). More specifically, our RQ1 shows that our LINE-DP achieves a more optimal predictive accuracy (i.e., a high recall with a reasonable number of false positives) than other techniques which not only produce few false positives but also achieve a low recall value. Our RQ2 shows that given the same amount of effort (20% of LOC), our LINE-DP can identify more defective lines (i.e., Top20%LOC recall) than these state-of-the-art techniques. Our RQ3 shows that our LINE-DP requires additional computation time of 3 seconds (8.46s - 5.74s) and 6 seconds (8.46s - 2.16s) compared to PMD and ErrorProne, respectively (see Figure 9b). These results suggest that based on the same amount of SQA effort, our LINE-DP can help developers identify more defective lines than the state-of-the-art techniques with small additional computation time. Thus, these findings highlight that our LINE-DP is a significant advancement for the development of line-level defect prediction in order to help practitioners prioritize the limited SQA resources in the most cost-effective manner.

Our LINE-DP depends on the performance of the file-level defect prediction model. The results of RQ2 and RQ3 are based on the file-level defect prediction models using the Logistic Regression technique. It is possible that our LINE-DP will miss defective lines if the file-level defect model misses defective files. In other words, the more accurate the file-level defect model is, the better the performance of our LINE-DP. Hence, improving the file-level defect model, e.g., optimizing the parameters [80] or using the advanced techniques (e.g., embedding techniques [62]), would improve the performance of our LINE-DP.

Recent studies have shown that deep learning and embedding techniques can improve the predictive accuracy of file-level defect models [6, 15, 62, 92]. However, the important features of the embedded source code identified by a model-agnostic technique cannot be directly mapped to the risky tokens. Hence, future work should investigate deep learning techniques to build accurate file-level models and/or techniques to utilize the embedded source code to identify risky tokens.

Our LINE-DP cannot identify defective lines that include only rare code tokens. During our manual categorization of RQ5, we observe that the defective lines that our LINE-DP has missed sometimes contain only tokens that rarely appear in the training dataset. This work uses a vector of token frequency as a feature vector to train the file-level model. Hence, future work should investigate an approach that can weight the important keywords that rarely appear in order to improve the predictive accuracy of our LINE-DP.

8.2 Threats to Validity

We now discuss possible threats to the validity of our empirical evaluation.

Construct Validity. It is possible that some defective lines are identified as clean when we construct the line-level defect datasets. In this work, we identify that bug-fixing commits are those commits that contain an ID of a bug report in the issue tracking system. However, some bug-fixing commits may not record such an ID of a bug report in the commit message. To ensure the quality of the dataset, we followed an approach suggested by prior work [13, 91], i.e., focusing on the issues that are reported after a studied release; labelled as bugs; affected only the studied release; and already closed or fixed. Nevertheless, additional approaches that improve the quality of the dataset (e.g., recovering missing defective lines) may further improve the accuracy of our results.

The chronological order of the data may impact the results of prediction models in the context of software vulnerability [41]. To address this concern, we use the defect datasets where defective files are labelled based on the affected version in the issue tracking system, instead of relying the assumption of a 6-month post-release window. In addition, we also perform an evaluation based on the cross-release setting which considers a time factor, i.e., using the past release ($k - 1$) to predict defects in the current release (k).

Internal Validity. The defect categorization of the qualitative analysis was mainly conducted by the first author. The result of manual categorization might be different when perform by others. To mitigate this threat, a subset of defect categorization results are verified by the other three authors of this paper. The level of agreement among the four coders is 0.86 and 0.81 for a subset of hit and missed defective blocks, respectively, indicating a perfect inter-rater agreement [86].

External Validation. The results of our experiment are limited to the 32 releases of nine studied software systems.

Future studies should experiment with other proprietary and open-source systems. To foster future replication of our study, we publish the benchmark line-level datasets.

9 Conclusions

In this paper, we propose a line-level defect prediction framework (called LINE-DP) to identify and prioritize defective lines to help developers effectively prioritize SQA effort. To the best of our knowledge, our work is the first to use the machine learning-based defect prediction models to predict defective lines by leveraging a state-of-the-art model-agnostic technique (called LIME). Through a case study of 32 releases of 9 software systems, our empirical results show that:

- Our LINE-DP achieves an overall predictive accuracy significantly better than the baseline approaches, with a median recall of 0.61 and 0.62 and a median false alarm of 0.47 and 0.48 for the within-release and cross-release settings, respectively.
- Given a fixed amount of effort (i.e., the top 20% of lines that are ranked by our LINE-DP), 26% and 27% of actual defective lines can be identified for the within-release and cross-release settings, respectively. On the other hand, only 17% - 22% of actual defective lines can be identified when ranking by the baseline approaches. Furthermore, fewer clean lines (false positives) will be examined to find the first defective line when ranking by our LINE-DP.
- The average computation time of our LINE-DP is 10.68 and 8.46 seconds for the within-release and cross-release settings, respectively. On the other hand, the baseline approaches take 0.89 to 26.85 seconds to identify defective lines.
- 63% of the defective lines that our LINE-DP can identify are categorized into the common defect types. More specifically, the majority defects that can be identified by our LINE-DP are related to argument change (32%) and condition change (18%).

The results show that our LINE-DP can effectively identify defective lines that contain common defects while requiring a smaller amount of SQA effort (in terms of lines of code) with a manageable computation time. Our work sheds the light on a novel aspect of leveraging the state-of-the-art model-agnostic technique (LIME) to identify defective lines, in addition to being used to explain the prediction of defective files from defect models [36]. Our framework will help developers effectively prioritize SQA effort.

Acknowledgement

C. Tantithamthavorn was partially supported by the Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme (DE200100941) and a Monash-FIT Early Career Researcher Seed Grant.

References

[1] B. Adams and S. McIntosh, "Modern release engineering in a nutshell–why researchers should care," in *SANER*, 2016, pp. 78–90.

[2] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan, "Building useful program analysis tools using an extensible java compiler," in *SCAM*, 2012, pp. 14–23.

[3] A. Agrawal, W. Fu, D. Chen, X. Shen, and T. Menzies, "How to "dodge" complex software analytics," *TSE*, vol. PP, 2019.

[4] A. Agrawal and T. Menzies, "Is "better data" better than "better data miners"?: on the benefits of tuning smote for defect prediction," in *ICSE*, 2018, pp. 1050–1061.

[5] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *FSE*, 2014, pp. 281–293.

[6] ——, "Suggesting accurate method and class names," in *FSE*, 2015, pp. 38–49.

[7] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *PACMPL*, vol. 3, 2019.

[8] D. Bowes, T. Hall, M. Harman, Y. Jia, F. Sarro, and F. Wu, "Mutation-Aware Fault Prediction," in *ISSTA*, 2016, pp. 330–341.

[9] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[10] Z. Chen, S. J. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *TSE*, 2019.

[11] J. Cohen, *Statistical power analysis for the behavioral sciences*. Routledge, 2013.

[12] T. Copeland, *PMD applied*. Centennial Books Arexandria, Va, USA, 2005, vol. 10.

[13] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-introducing Changes," *TSE*, vol. 43, no. 7, pp. 641–657, 2017.

[14] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, "Lessons learned from using a deep tree-based model for software defect prediction in practice," in *MSR*, 2019, p. 46–57.

[15] H. K. Dam, T. Tran, T. T. M. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for predicting vulnerable software components," *TSE*, 2018.

[16] M. D'Ambros, M. Lanza, and R. Robbes, "An Extensive Comparison of Bug Prediction Approaches," in *MSR*, 2010, pp. 31–41.

[17] E. Engström, P. Runeson, and G. Wikstrand, "An empirical evaluation of regression testing based on fix-cache recommendations," in *ICST*, 2010, pp. 75–78.

[18] A. Field, *Discovering statistics using IBM SPSS statistics*. sage, 2013.

[19] D. Fisher, R. DeLine, M. Czerwinski, and S. Drucker, "Interactions with big data analytics," *Interactions*, vol. 19, no. 3, pp. 50–59, 2012.

[20] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," *SIGPLAN Notices*, vol. 37, no. 5, p. 234–245, 2002.

[21] W. Fu, T. Menzies, and X. Shen, "Tuning for Software Analytics: Is it really necessary?" *IST*, vol. 76, pp. 135–146, 2016.

[22] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting

the Impact of Classification Techniques on the Performance of Defect Prediction Models," in *ICSE*, 2015, pp. 789–800.

[23] ——, "A Large-scale Study of the Impact of Feature Selection Techniques on Defect Classification Models," in *MSR*, 2017, pp. 146–157.

[24] G. Gousios, M. Pinzger, and A. v. Deursen, "An Exploratory Study of the Pull-based Software Development Model," in *ICSE*, 2014, pp. 345–355.

[25] A. Habib and M. Pradel, "How many of all bugs do we find? a study of static bug detectors," in *ASE*, 2018, pp. 317–328.

[26] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A Systematic Literature Review on Fault Prediction Performance in Software Engineering," *TSE*, vol. 38, no. 6, pp. 1276–1304, 2012.

[27] F. E. Harrell Jr, *Regression Modeling Strategies : With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis*. Springer, 2015.

[28] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *ICSE*, 2012, pp. 200–210.

[29] S. Heckman and L. Williams, "On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques," in *ESEM*, 2008, pp. 41–50.

[30] S. S. Heckman, "Adaptively ranking alerts generated from automated static analysis," *XRDS: Crossroads, The ACM Magazine for Students*, vol. 14, no. 1, p. 7, 2007.

[31] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *FSE*, 2017, pp. 763–773.

[32] K. Herzig, "Using pre-release test failures to build early post-release defect prediction models," in *ISSRE*, 2014, pp. 300–311.

[33] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *ICSE*, 2012, pp. 837–847.

[34] Q. Huang, X. Xia, and D. Lo, "Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction," in *ICSME*, 2017, pp. 159–170.

[35] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *ASE*, 2013, pp. 279–289.

[36] J. Jiarpakdee, C. Tantithamthavorn, H. K. Dam, and J. Grundy, "An empirical study of model-agnostic techniques for defect prediction models," *TSE*, 2020.

[37] J. Jiarpakdee, C. Tantithamthavorn, and A. E. Hassan, "The Impact of Correlated Metrics on Defect Models," *TSE*, 2019.

[38] J. Jiarpakdee, C. Tantithamthavorn, A. Ihara, and K. Matsumoto, "A Study of Redundant Metrics in Defect Prediction Datasets," in *ISSREW*, 2016, pp. 51–52.

[39] J. Jiarpakdee, C. Tantithamthavorn, and C. Treude, "AutoSpearman: Automatically Mitigating Correlated Software Metrics for Interpreting Defect Models," in *ICSME*, 2018, pp. 92–103.

[40] ——, "The impact of automated feature selection techniques on the interpretation of defect models," *EMSE*, 2020.

[41] M. Jimenez, R. Rwemalika, M. Papadakis, F. Sarro, Y. Le Traon, and M. Harman, "The importance of accounting for real-world labelling when predicting software vulnerabilities," in *ESEC/FSE*, 2019, pp. 695–705.

[42] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *ICSE*, 2013, pp. 672–681.

[43] 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 *ICSME*, 2010, pp. 1–10.

[44] 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 *ESEM*, 2007, pp. 196–204.

[45] 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," *TSE*, vol. 39, no. 6, pp. 757–773, 2013.

[46] M. Kim, J. Nam, J. Yeon, S. Choi, and S. Kim, "REMI: Defect prediction for efficient API testing," in *ESEC/FSE*, 2015, pp. 990–993.

[47] S. Kim and M. D. Ernst, "Which warnings should i fix first?" in *ESEC/FSE*, 2007, pp. 45–54.

[48] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation exploitation in error ranking," in *FSE*, 2004, p. 83–93.

[49] T. Kremenek and D. Engler, "Z-ranking: Using statistical analysis to counter the impact of static analysis approximations," in *Static Analysis*, 2003, pp. 295–315.

[50] Lucidchart, *Release management process*, 2020 (accessed July 23, 2020), https://www.lucidchart.com/blog/release-management-process.

[51] G. Macbeth, E. Razumiejczyk, and R. D. Ledesma, "Cliff's Delta Calculator: A Non-parametric Effect Size Program for Two Groups of Observations," *Universitas Psychologica*, vol. 10, pp. 545–555, 2011.

[52] L. Massaron and A. Boschetti, *Regression Analysis with Python*. Packt Publishing Ltd, 2016.

[53] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *CSMR*, 2010, pp. 107–116.

[54] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with precision: A response to" comments on'data mining static code attributes to learn defect predictors'"," *TSE*, vol. 33, no. 9, pp. 637–640, 2007.

[55] T. Menzies, J. Greenwald, and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," *TSE*, vol. 33, no. 1, pp. 2–13, 2007.

[56] T. Menzies and M. Shepperd, ""bad smells" in software analytics papers," *IST*, vol. 112, pp. 35 – 47, 2019.

[57] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *ICSE*, 2005, pp. 580–586.

[58] ——, "Use of Relative Code Churn Measures to Predict System Defect Density," *ICSE*, pp. 284–292, 2005.

[59] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change Bursts as Defect Predictors," in *ISSRE*, 2010, pp. 309–318.

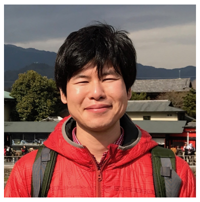
[60] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *ISSTA*, 2011, pp. 199–209.

[61] L. Pascarella, F. Palomba, and A. Bacchelli, “Fine-grained just-in-time defect prediction,” *JSS*, vol. 150, pp. 22–36, 2019.
[62] M. Pradel and K. Sen, “Deepbugs: A learning approach to name-based bug detection,” *PACMPL*, vol. 2, p. 147, 2018.
[63] F. Rahman and P. Devanbu, “How, and Why, Process Metrics are Better,” in *ICSE*, 2013, pp. 432–441.
[64] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu, “Comparing static bug finders and statistical prediction,” in *ICSE*, 2014, pp. 424–434.
[65] M. T. Rahman and P. C. Rigby, “Release stabilization on linux and chrome,” *IEEE Software*, vol. 32, no. 2, pp. 81–88, 2015.
[66] M. Rahman, D. Palani, and P. C. Rigby, “Natural software revisited,” in *ICSE*, 2019, pp. 37–48.
[67] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, “On the Naturalness of Buggy Code,” in *ICSE*, 2016, pp. 428–439.
[68] V. Raychev, M. Vechev, and E. Yahav, “Code completion with statistical language models,” *ACM Sigplan Notices*, vol. 49, no. 6, pp. 419–428, 2014.
[69] M. T. Ribeiro, S. Singh, and C. Guestrin, “”why should I trust you?”: Explaining the predictions of any classifier,” in *SIGKDD*, 2016, pp. 1135–1144.
[70] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, “Predicting accurate and actionable static analysis warnings: an experimental approach,” in *ICSE*, 2008, pp. 341–350.
[71] M. Shepperd, D. Bowes, and T. Hall, “Researcher Bias: The Use of Machine Learning in Software Defect Prediction,” *TSE*, vol. 40, no. 6, pp. 603–616, 2014.
[72] D. Spadini, M. Aniche, and A. Bacchelli, “Pydriller: Python framework for mining software repositories,” in *ESEC/FSE*, 2018, pp. 908–911.
[73] C. Tantithamthavorn, “Towards a Better Understanding of the Impact of Experimental Components on Defect Prediction Modelling,” in *ICSE-DS*, 2016, pp. 867–870.
[74] C. Tantithamthavorn and A. E. Hassan, “An Experience Report on Defect Modelling in Practice: Pitfalls and Challenges,” in *ICSE-SEIP*, 2018, pp. 286–295.
[75] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, “The Impact of Class Rebalancing Techniques on The Performance and Interpretation of Defect Prediction Models,” *TSE*, 2019.
[76] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto, “The Impact of Mislabelling on the Performance and Interpretation of Defect Prediction Models,” in *ICSE*, 2015, pp. 812–823.
[77] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, “Automated Parameter Optimization of Classification Techniques for Defect Prediction Models,” in *ICSE*, 2016, pp. 321–332.
[78] ——, “Comments on “Researcher Bias: The Use of Machine Learning in Software Defect Prediction”,” *TSE*, vol. 42, no. 11, pp. 1092–1094, 2016.
[79] ——, “An Empirical Comparison of Model Validation Techniques for Defect Prediction Models,” *TSE*, vol. 43, no. 1, pp. 1–18, 2017.
[80] ——, “The Impact of Automated Parameter Optimization on Defect Prediction Models,” *TSE*, 2018.
[81] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, “Investigating code review practices in defective files: An empirical study of the qt system,” in *MSR*, 2015, p. 168–179.
[82] ——, “Revisiting Code Ownership and its Relationship with Software Quality in the Scope of Modern Code Review,” in *ICSE*, 2016, pp. 1039–1050.
[83] M. Tomczak and E. Tomczak, “The need to report effect size estimates revisited. An overview of some recommended measures of effect size.” *Trends in Sport Sciences*, vol. 21, no. 1, pp. 19–25, 2014.
[84] Z. Tu, Z. Su, and P. Devanbu, “On the localness of software,” in *FSE*, 2014, pp. 269–280.
[85] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, “How developers engage with static analysis tools in different contexts,” *EMSE*, 2019.
[86] A. J. Viera and J. M. Garrett, “Understanding interobserver agreement: the kappa statistic,” *Family Medicine*, vol. 37, pp. 360–363, 2005.
[87] Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin, and X. Yang, “Perceptions, expectations, and challenges in defect prediction,” *TSE*, 2018.
[88] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, “Bugram: Bug detection with n-gram language models,” in *ASE*, 2016, pp. 708–719.
[89] S. Wang, T. Liu, J. Nam, and L. Tan, “Deep Semantic Feature Learning for Software Defect Prediction,” *TSE*, 2018.
[90] S. Wang, T. Liu, and L. Tan, “Automatically Learning Semantic Features for Defect Prediction,” in *ICSE*, 2016, pp. 297–308.
[91] S. Yathish, J. Jiarpakdee, P. Thongtanunam, and C. Tantithamthavorn, “Mining Software Defects: Should We Consider Affected Releases?” in *ICSE*, 2019, pp. 654–665.
[92] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, “A novel neural source code representation based on abstract syntax tree,” in *ICSE*, 2019, pp. 783–794.

Supatsara Wattanakriengkrai is a Master’s student at the Department of Information Science, Nara Institute of Science and Technology, Japan. She received her BS degree in Information and Communication Technology from Mahidol University, Thailand, in 2019. Her main research interests are Empirical Software Engineering, Mining Software Repositories, and Software Quality Assurance. Contact her at wattanakri.supatsara.ws3@is.naist.jp.

Patanamon Thongtanunam is a lecturer at the School of Computing and Information Systems, the University of Melbourne, Australia. She received PhD in Information Science from Nara Institute of Science and Technology, Japan. Her research interests include empirical software engineering, mining software repositories, software quality, and human aspect. Her research has been published at top-tier software engineering venues like International Conference on Software Engineering (ICSE) and Journal of Transaction on Software Engineering (TSE). More about Patanamon and her work is available online at http://patanamon.com.

Chakkrit Tantithamthavorn is a 2020 ARC DECRA Fellow and a Lecturer in Software Engineering in the Faculty of Information Technology, Monash University, Melbourne, Australia. His current fellowship is focusing on the development of “Practical and Explainable Analytics to Prevent Future Software Defects”. His work has been published at several top-tier software engineering venues, such as the IEEE Transactions on Software Engineering (TSE), the Springer Journal of Empirical Software Engineering (EMSE) and the International Conference on Software Engineering (ICSE). More about Chakkrit and his work is available online at http://chakkrit.com.



Hideaki Hata is an assistant professor at the Nara Institute of Science and Technology. His research interests include software ecosystems, human capital in software engineering, and software economics. He received a Ph.D. in information science from Osaka University. More about Hideaki and his work is available online at https://hideakihata.github.io/.

Kenichi Matsumoto is a professor in the Graduate School of Science and Technology at the Nara Institute of Science and Technology. His research interests include software measurement and software processes. He received a Ph.D. in information and computer sciences from Osaka University. He is a Senior Member of the IEEE and a member of the IEICE and the IPSJ. Contact him at matumoto@is.naist.jp.