Just-In-Time Defect Identification and Localization: A Two-Phase Framework

Meng Yan, Xin Xia, Yuanrui Fan, Ahmed E. Hassan, David Lo, Shanping Li

Abstract—Defect localization aims to locate buggy program elements (e.g., buggy files, methods or lines of code) based on defect symptoms, e.g., bug reports or program spectrum. However, when we receive the defect symptoms, the defect has been exposed and negative impacts have been introduced. Thus, one challenging task is: whether we can locate buggy program prior to the appearance of the defect symptom (e.g., when buggy program elements are being committed to a version control system). We refer to this type of defect localization as **"Just-In-Time (JIT) Defect localization"**. Although many prior studies have proposed various JIT defect identification methods to identify whether a new change is buggy, these prior methods do not locate the suspicious positions. Thus, JIT defect localization is the next step of JIT defect identification (i.e., after a buggy change is identified, suspicious source code lines are located).

To address this problem, we propose a two-phase framework, i.e., JIT defect identification and JIT defect localization. Given a new change, JIT defect identification will identify it as buggy change or clean change first. If a new change is identified as buggy, JIT defect localization will rank the source code lines introduced by the new change according to their suspiciousness scores. The source code lines ranked at the top of the list are estimated as the defect location. For JIT defect identification phase, we use 14 change-level features to build a classifier by following existing approach. For JIT defect localization phase, we propose a JIT defect localization approach that leverages software naturalness with the N-gram model. To evaluate the proposed framework, we conduct an empirical study on 14 open source projects with a total of 177,250 changes. The results show that software naturalness is effective for our JIT defect localization. Our model achieves a reasonable performance, and outperforms the two baselines (i.e., random guess and a static bug finder (i.e., PMD)) by a substantial margin in terms of four ranking measures.

Index Terms—Defect Localization, Just-in-Time, Defect Identification, Software Naturalness

1 Introduction

During software development and maintenance, developers often spend much effort and resources to debug a program [1]. Defect localization (or bug localization) aims to help developers to locate buggy program elements, such as buggy files, methods or lines of code. Researchers have proposed various techniques for defect localization, including information retrieval (IR) based techniques [2]–[5] and spectrum-based techniques [6]–[8]. These techniques perform localization by analyzing the defect symptoms. These symptoms could be a description of a bug, or a failing test case. For example, IR based techniques analyze the textual description in bug reports, spectrum-based techniques analyze program spectrum of failing and successful execution traces [1].

However, one main limitation of the above-mentioned localization techniques is that they rely on defect symptoms (i.e., from bug reports or execution traces). When a defect symptom is discovered, the defect has already been exposed and it has already negatively impacted the software. Therefore, one challenging task is: can we locate buggy program prior to the appearance of the defect symptom (e.g., when buggy program elements are being committed to a version control system)? We refer to this type of defect localization as **"Just-In-Time (JIT) Defect Localization"**.

The idea of "JIT Defect Localization" comes from "JIT Defect Identification" (aka, JIT defect prediction) which is a well-known technique for identifying buggy (i.e., defect-introducing) changes at check-in time. Recently, JIT defect identification has attracted an increasing research interest, a number of studies have proposed various techniques for JIT defect identification [9]–[16]. JIT defect identification can yield many benefits. For example, the identification can be performed at the time when the change is submitted; such immediate feedback ensures that the context is still fresh in the minds of developers. This fresh context can speed up the fixing of the buggy change. Additionally, the identification is made at a fine-granularity, i.e., change-level, that are mapped to a few areas of the large code base. Such a fine-granularity identification can provide large effort saving over coarser grained identification (e.g., file-level or module-level) [17].

Despite the achievements of JIT defect identification,

- *Meng Yan is with School of Big Data and Software Engineering, Chongqing University, Chongqing, China and the College of Computer Science and Technology, Zhejiang University, Hangzhou, China and PengCheng Laboratory.*
  E-mail: mengy@cqu.edu.cn
- *Xin Xia is with the Faculty of Information Technology, Monash University, Melbourne, Australia.*
  E-mail: xin.xia@monash.edu
- *Yuanrui Fan and Shanping Li are with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China.*
  E-mail: {yrfan, shan}@zju.edu.cn
- *Ahmed E. Hassan is with the School of Computing, Queen's University, Canada.*
  E-mail: ahmed@cs.queensu.ca
- *David Lo is with the School of Information Systems, Singapore Management University, Singapore.*
  E-mail: davidlo@smu.edu.sg
- *Xin Xia is the corresponding author.*

Manuscript received ; revised

one issue remains unanswered is: what is the next step if we identified a buggy change? Our main concern is that all of the prior studies focus on how to classify a change (i.e., classify a change as buggy or clean), or sort changes according to their defective likelihood scores, to find more buggy changes by inspecting fewer changed lines of code (i.e., effort-aware model) [9], [12], [15], [16]. However, after a buggy change is identified, it is also a difficult task to locate the exact buggy positions (e.g., source code lines), especially for a change that has added many lines of code (LOC). Take the Jmeter project as an example: the average newly added LOC across all the changes is 180. It would cost a great amount of inspection effort if we inspect all the added LOC for all the buggy changes. Unfortunately, there is no technique for guiding how to inspect a buggy change in order to locate the exact buggy lines with less inspection effort.

Therefore, in response to the concerns raised above, we highlight the framework of JIT Defect Identification and Localization. Different from current defect localization techniques by analyzing defect symptoms, JIT defect localization aims to be an early step of continuous quality control because it can be performed as soon as a code change is checked in. In summary, the main benefits of performing JIT defect identification and localization are as follows:

- *Localization is performed at a fine-granularity.* JIT defect localization locates buggy program elements at a fine-granularity, i.e., line-level. For each buggy change, JIT defect localization locates the buggy lines in a buggy change. Such a fine-granularity can help developers to locate and address defects using less effort.
- *Localization is performed early on.* JIT defect localization is performed at check-in time. Such immediate feedback ensures that the context is still fresh in the minds of developers. This fresh context can help speed up the fixing of located defects immediately.
- *Localization is performed without relying on defect symptoms.* In JIT defect localization, only change properties are needed. Thus, the localization can be invoked without analyzing the defect symptoms. In contrast to current defect localization techniques which require the analysis of defect symptoms, the benefit of JIT defect localization is that it can locate the defect before the appearance of its symptom.

The basic technical idea of our JIT localization technique begins with the "software naturalness" as observed by Hindle et al [18] who noted that "natural" code is highly repetitive and can be captured through a language model (e.g. a N-gram model that was originally developed in Natural Language Processing field). Based on this observation, Ray et al. [19] observed that buggy code tends to be more entropic (i.e. unnatural) compared with clean code. Moreover, they observed that this observation can be helpful for defect identification on release level and can complement the effectiveness of static bug finders (e.g., FindBugs and PMD).

Inspired by the above-mentioned observations, we propose an automated two-phase framework. The **first phase** consists of a JIT defect identification (i.e., identify buggy changes). The **second phase** consists of a JIT defect localization for the buggy changes that were correctly identified by the first phase. The first phase is similar to the prior approaches. We implement a JIT defect identification approach by following prior studies [9], [15], [20]. In the second phase, we propose a localization approach based on software naturalness by learning from historical labeled code.

In summary, our work consists of three steps as shown in Figure 1. (1) The first step is a data preparation step. We identify buggy and clean changes using the Refactoring Aware SZZ (RA-SZZ) algorithm first [21], [22], then we link each buggy change and its corresponding bug-fixing change(s). Subsequently, we label the added lines by buggy changes that are modified by the corresponding bug-fixing changes as buggy code; otherwise, the lines are labeled as clean code. (2) The second step is the model building phase. We build a JIT defect identification model (i.e., JIT Defect Identifier) and a JIT defect localization model (i.e., JIT Defect Locator). For the JIT defect identifier, we use 14 change-level features to build a logistic classifier by following prior studies [15], [20]. For the JIT defect locator, we build a code language model based on historical clean code using the N-gram model that has shown to be an effective technique for source code modeling [18], [19], [23], [24]. (3) The third step is the model application phase. For a new change, we first identify whether or not it is a likely buggy change using our JIT defect identifier. For a likely buggy change, we sort the added lines according to the likelihood values using the JIT defect locator. The lines sorted at the top are more likely to be the defect location.

1.1 Novelty Statement

The novelty of this paper is to propose a new framework: Just-In-Time Defect Identification and Localization. The proposed framework contains a two-phase analysis, i.e., (1) identifying buggy changes and (2) locating suspicious code lines introduced in the identified buggy change.

1.2 Contributions

The contributions of this paper are as follows:

- We propose a two-phase framework of JIT defect identification and localization. This framework can locate suspicious defective lines on change-level at check-in time.
- We conduct an empirical study to evaluate our proposed framework on 14 projects with a total of 177,250 changes. The results show that our framework achieves a reasonable performance, which significantly outperforms the baselines (i.e., random guess and a static bug finder (i.e., PMD)) by a substantial margin[1].

1. Our replication package: https://github.com/MengYan1989/JIT-DIL

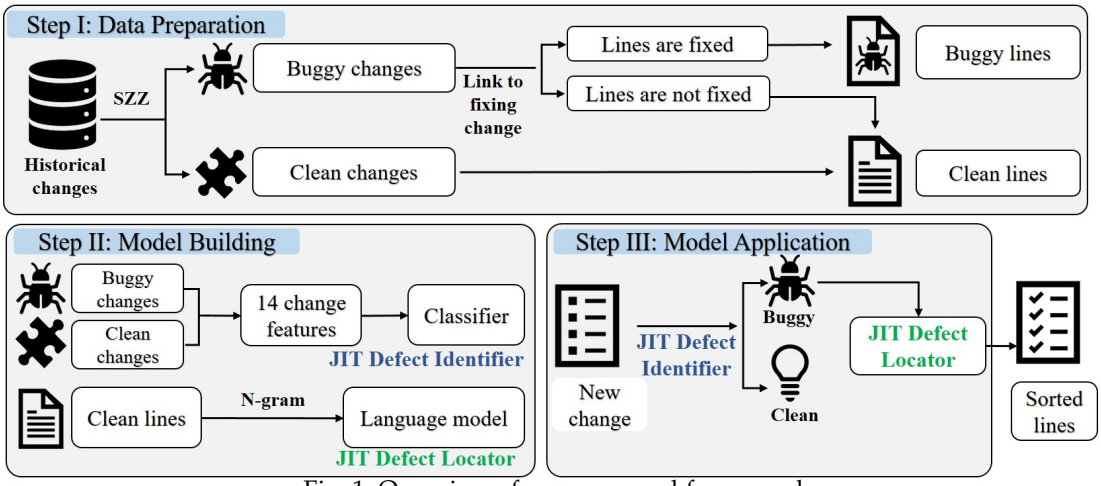


Fig. 1: Overview of our proposed framework

1.3 Paper Organization

The rest of the paper is structured as follows. Section 2 presents the details of our proposed framework. Section 3 provides the empirical study setup, including the dataset, validation setting and model evaluation. In Section 4, we provide the experimental results and their analysis. Section 5 presents the discussions on the impact of different configurations. Section 6 describes the threats to validity. Section 7 presents the related work of our study. Section 8 concludes and presents future work.

2 Our Framework

Figure 1 presents an overview of our proposed framework. Our framework consists of three steps: data preparation, model building and model application. In this section, we first describe how to prepare the dataset to build our model. Then we provide the details about how to build our model. Finally, we present how to apply our framework to JIT defect localization.

2.1 Data Preparation

In this phase, we identify and collect clean and buggy source code lines that are added by software changes. Clean lines are used to build a "Clean" language model, buggy lines are used to be the ground truth for evaluating the localization performance. We focus on added lines by software change because our model aims to perform the localization on the added lines of new changes. Note that a modified line of a change is treated as an added line (after modification) and a deleted line (before modification).

Similarly to Ray et al. [19], we apply the SZZ algorithm [25] to identify clean and buggy lines. Many prior studies reported that Śliwerski et al.'s original SZZ algorithm [25] is impacted by large amounts of noise (e.g., blank/comment lines) [21], [22], [26]–[28]. Following prior study's recommendation [22], we apply Refactoring Aware SZZ (RA-SZZ) [21]—an SZZ variant that can deal with noise including blank/comment lines, format modifications and refactoring modifications.

In summary, we take four steps to prepare the studied data:

1) **Defect-fixing change identification.** We identify the changes that fix defects following prior work [22]. For each change, we first search the commit message for references to issue reports, e.g., "Fixed #233". Then, we crawl the corresponding issue report from the issue tracking system (ITS) of the project and check whether the report is defined as a defect in the ITS. If the report is defined as a defect and it is resolved or closed, we consider the change as a defect-fixing change.

2) **Buggy (i.e., defect-introducing) change identification.** We leverage RA-SZZ[2] [21], [22] to identify defect-introducing changes. RA-SZZ first leverages the `git diff` command to identify the lines that were changed by defect-fixing changes. From the identified lines, RA-SZZ filters away blank/comment lines, lines involving format modifications (e.g., modification of code indentation) and those involving refactoring modifications. Then, for the remaining lines, RA-SZZ traces back through the change history to identify the changes that introduce the lines, which are identified as defect-introducing changes. RA-SZZ leverages the annotation graph [29] to trace back through the change history. Annotation graph was proposed by Zimmermann et al. [29]. The graph traces the evolution of lines of code along the code change history. Using the annotation graph, RA-SZZ will not stop its search for defect introduction changes at changes that involve format modifications or refactoring modifications instead it can further trace back through the code history to identify the actual defect-introducing changes. In addition, another advantage of RA-SZZ is that for a code statement involving multiple lines, RA-SZZ can automatically combine the lines into one line. Hence, RA-SZZ ensures the completeness of the code lines that are analyzed in our study.

3) **Buggy and clean lines identification.** We label the lines that were added by clean changes, and lines that were added by buggy changes but were not later fixed, as clean lines. And we label the lines that were added by buggy changes, and were later fixed by linked defect-fixing changes as buggy lines. Since we aim to analyze the source code to locate likely defective code lines, the source code comments are out of the scope of our consideration.

2. https://github.com/danielcalencar/ra-szz

Therefore, we remove all the source code comments in the added lines using regular expression.

4) **Code tokenization.** After collecting buggy and clean code lines, we perform the code tokenization step to break each line into separate words. Specifically, we use the tokenization tool that delimit code based on the Java grammar proposed by prior study [24]. For example, a line in Deeplearning4J project is *"System.out.println(graph.summary(InputType.feedForward(5)));"*, the tokenized words are: "`System . out . println ( graph . summary ( InputType . feedForward ( 5 ) ) ) ;`". All the tokenized words of code lines are included for building a code language model.

2.2 Model Building

The model building step consists of two models, i.e., the JIT defect identification model and the JIT defect localization model.

1) **JIT defect identification model.**

Our JIT defect identification model aims to identify whether or not a new change is a buggy change. To do so, we implemented a logistic regression based approach as done by prior studies [9], [15]. We briefly introduce the steps of the JIT defect identification below, and more details can be found in the original papers [9], [15]. First, we extract 14 change-level features proposed by prior studies [9], [12], [15], [16], [30] and presented in Table 1. Second, we do the same data preprocessing steps (e.g. re-sampling and log transformation) as prior studies [9], [15]. Third, since we already labeled each historical change as buggy or clean in our data preparation step, we build the identification model using a logistic regression classifier by training on historical changes. Subsequently, a new change would be classified as buggy if its predicted likelihood is larger than 0.5; otherwise it will be identified as clean.

Table 1 presents the name and description of our used 14 change-level features. These features are grouped into five dimensions: diffusion (NS, ND, NF and Entropy), size (LA, LD and LT), purpose (FIX), history (NDEV, AGE and NUC) and experience (EXP, REXP and SEXP). The diffusion dimension characterizes the distribution of a change. The purpose dimension only consists of FIX, which indicates whether or not this commit is a bug fixing commit. The history dimension characterizes how developers modify the files within the change in the code history. The experience dimension captures a developer's experience based on the number of commits made by that particular developer in the past. We calculate these features following prior studies [9], [15], more detailed description can be found at Kamei et al.'s work [9].

2) **JIT defect localization model.**

To build a localization model, we build a source code language model trained on clean source code lines (i.e., a clean model) by using N-gram model.

N-gram modeling for source code. In our work, we choose N-gram model as our underlying language modeling technique because N-gram model has been shown to be effective in modeling source code [18], [19], [23], [24]. Our objective is to propose the first JIT defect localization framework. Thus, we believe that a simple and popular modeling technique is sufficient.

TABLE 1: Summary of the used change features.

Dimension	Feature	Description
Diffusion	NS	Number of subsystems touched by the current change
	ND	Number of directories touched by the current change
	NF	Number of files touched by the current change
	Entropy	Distribution across the touched files
Size	LA	Lines of code added by the current change
	LD	Lines of code deleted by the current change
	LT	Lines of code in a file before the current change
Purpose	FIX	Whether or not the current change is a defect fix
History	NDEV	Number of developers that changed the files
	AGE	Average time interval between the last and current change
	NUC	Number of unique changes to the files
Experience	EXP	Developers experience
	REXP	Recent developer experience
	SEXP	Developer experience on a subsystem

Formally, a language model assigns a probability (or a score) to a sequence of words. In our context, suppose there is a code fragment s of length $|s|$, it is tokenized into t_1, t_2,...,t_s. A language model estimates the probability of this sequence occurring as a product of a series of conditional probabilities as:

$$p(s) = \prod_{i=1}^{|s|} p(t_i|t_1, ..., t_{i-1}). \quad (1)$$

Specifically, $p(t_i|t_1, ..., t_{i-1})$ denotes the probability that the token t_i follows the previous tokens, i.e., the prefix $h = t_1, ..., t_{i-1}$. The count-based MLE *(count of sequence/count of context)* that is used in N-gram models becomes impractical to estimate the probabilities, due to the very large number of possible prefixes. Thus, a common optimization is the N-gram language model. The N-gram model assigns a probability (or a score) to a sequence of words based on the Markov-assumption, i.e., each token is conditioned on the $N-1$ preceding tokens, that is,

$$p(t_i|h) = p(t_i|t_{i-n+1}, ..., t_{i-1}). \quad (2)$$

We estimate the above-mentioned probability from the training corpus as the fraction of times that t_i follows the prefix $t_{i-n+1}, ..., t_{i-1}$. Additionally, since the probabilities may vary by orders of magnitude, we use the (typically negated) logarithm of the phrase probability, to arrive at the information-theoretic measure of entropy as defined in prior studies [19], [24]. Entropy reflects the number of needed bits to encode the phrase (and, analogously, a token) given the language model. Entropy is a measure of how "surprised" a model is by the given document. The lower the entropy of a new code fragment is, the more natural the new code fragment is with the training code corpus. Formally, given a code fragment s of length $|s|$, the prefix of each token is denoted by $h = t_1, ..., t_{i-1}$, the entropy of the code fragment (and, analogously, a token) is computed as:

$$H_p(s) = -\frac{1}{|s|} \log_2 p(s) = -\frac{1}{|s|} \sum_{i=1}^{|s|} \log_2 p(t_i|h). \quad (3)$$

Note that the model training corpus only consists of the added clean lines by software changes that are identified in the data preparation step. The vocabulary is built from the training corpus. The reason is that we attempt to build a pure-clean training corpus and our target is line-level localization. Adding more context (e.g., other lines before/after a clean line) might impact the cleanliness of our training corpus.

N-gram configuration. By default, in terms of the N-gram length (i.e., the value size N in an N-gram model), we set $N = 6$ (i.e., 6-gram) since prior study by Hellendoorn and Devanbu [24] has shown that the value performs well for modeling source code. Additionally, in practice, it is necessary to smooth the probability distributions by assigning non-zero probabilities to unseen words or N-grams. The smoothing is needed since models that are derived directly from the n-gram frequency counts have severe problems when confronted with any n-grams that have not explicitly been seen before, i.e., the zero-frequency problem. Smoothing is a popular technique to cope with this problem in the NLP field [31]. Various smoothing methods have been proposed, such as Jelinek-Mercer (JM), Witten-Bell (WB) and Absolute Discounting (AD) [24], [31]. By default, we adopt the JM smoothing method that has shown to perform well for modeling source code as recommended by Hellendoorn and Devanbu [24].

2.3 Model Application

JIT defect identification. Given a new change, the built JIT defect identification model in previous subsection will identify it as buggy or clean.

JIT defect localization. For the likely buggy changes identified in previous step, the built JIT defect localization model will computes the entropy of each token, then we compute the entropy of a line using the entropy values of all its tokens. Subsequently, we sort all the introduced lines in descending order according the entropy of each line. Lines that are sorted at the top are more likely to be defective.

One final issue that remains is how to compute the line entropy (i.e., the entropy-based score of a line) according to the entropy values of its tokens. Suppose there is a code line s of length $|s|$, the line is tokenized into t_1, t_2,...,$t_{|s|}$. The entropy of each token outputted by language model is denoted as $H_p(t_1)$, $H_p(t_2)$,...,$H_p(t_{|s|})$. In a prior study [19], the line entropy $H_p(s)$ is computed by simply averaging the entropy values of its tokens, that is: $H_p(s) = \frac{1}{|s|}\sum_{i=1}^{|s|} H_p(t_i)$.

In our context, since our model aims to sort lines according to line entropy to find more likely buggy lines, we conjecture that the most unnatural token sequences is important for sorting. Therefore, we adjust line entropy $H_p(s)$ computing method by combining maximum entropy and the average entropy.

Specifically, in terms of the clean model, we compute the line entropy (denoted as $H_p(s)_c$) by summing the maximum (i.e., max) entropy with the average entropy of its tokens, that is,

$$H_p(s)_c = max(H_p(t_1), ..., H_p(t_{|s|})) + \frac{1}{|s|}\sum_{i=1}^{|s|} H_p(t_i). \quad (4)$$

TABLE 2: Summary of the studied projects.

Project	Start	#Changes	#Buggy	Ratio
Deeplearning4j	Nov-13	8,770	2,497	0.28
Jmeter	Sep-98	14,625	1,542	0.11
H2o	Mar-14	21,914	1,047	0.05
Libgdx	Mar-10	13,019	2,042	0.16
Jetty	Mar-09	14,804	923	0.06
Robolectric	Jun-10	7,085	697	0.10
Storm	Sep-11	8,819	401	0.05
Jitsi	Jul-05	12,608	824	0.07
Jenkins	Nov-06	23,764	1,436	0.06
Graylog2-server	May-10	13,702	1,466	0.11
Flink	Dec-10	11,982	1,184	0.10
Druid	May-11	5,417	1,068	0.20
Closure-compiler	Nov-09	10,870	249	0.02
Activemq	Dec-05	9,871	1,738	0.18

The reasons for this adjustment are: 1) the max entropy captures the most unnatural token sequences of a line. Within the context of the clean model, the most unnatural line means the most defective line. 2) The average entropy captures the entire naturalness of a line, summing it with max entropy is useful for describing the entire naturalness of the line, especially when the max entropy of different lines might be equal.

3 Empirical Study Setup

In this section, we present our empirical study setup. First, we present a summary of the used dataset. Second, we present our validation setting to create the training and testing sets for our study. Third, we describe the performance measures that are used to evaluate the JIT defect localization method. At last, we present the research questions that we are interested in answering in our empirical study.

3.1 Dataset

We select experimental projects from Github. The selected projects are written by Java, cover different application domains, are of different sizes and have a varying number of contributors. Additionally, all the projects have over 5,000 changes to ensure sufficient samples for modeling, and over 1,000 stars to ensure that the studied projects are non-trivial ones [32]. We randomly selected 14 projects that satisfy our inclusion criteria. The sample size is larger than the related JIT studies [9], [12], [15], [16], [20]. Table 2 presents the summary of studied projects.

We collect the changes of studied projects from the creation data of the projects to March 1, 2018. However, since we need to use the future changes to identify defect-introducing changes, we use the changes until October 1, 2017 (i.e., a five-month window) to ensure most of the studied changes are correctly labeled. We set the five-month window because above 80% of the buggy changes in our dataset were fixed within five months on average. There are a few buggy changes that may need even over a year to be fixed. A much larger window may reduce the number of instances for our study. A shorter time window would likely introduce noise in our data. Thus, we think five months is a rational and sufficient window. In total, there are 177,250 changes in the studied projects.

3.2 Model Evaluation

Evaluation setting. Our proposed framework aims to locate the buggy lines that are introduced by a **buggy change**. Thus, the JIT defect localization phase is only useful when a buggy change is identified in JIT defect identification phase. Under this situation, we report the evaluation results considering two kinds of settings. One setting is the evaluation of *identified-buggy* changes in the testing set. This setting means that we only consider the buggy changes that are correctly identified by our first phase (i.e., the JIT defect identification phase). This setting aims to simulate the practical usage scenario when using our tool. The other setting is the evaluation of *all-buggy* changes in testing set. This setting assumes that we already know whether or not a change is a buggy one, then we apply our localization approach (i.e., supposing we have a perfect JIT defect identification approach). In our evaluation, we report the results of these two settings.

Performance measures. ***MRR*** and ***MAP*** are classical evaluation metrics for information retrieval [33]. In our context, MRR measures how far we need to check in down a sorted list of added lines of a buggy change to locate the first buggy line, while MAP considers the ranks of all buggy lines in that sorted list. MRR and MAP are widely used in many software engineering studies to evaluate the performance of a retrieval task [34]–[37]. Note that our evaluation is performed at the change-level, i.e., each buggy change in our test set has an MRR and an MAP performance value. And we report the average of *identified-buggy* and *all-buggy* changes in the testing set in a project.

Top-k Accuracy. In practice, inspecting all changed LOC in all changes is probably unrealistic. Prior studies assumed that only a small proportion of the changed LOC could be inspected given the limited resources and time in practice [19], [38]. Therefore, we also evaluated our approach considering a limited inspection budget (i.e., inspecting the most Top-k buggy lines of a change).

Top-k accuracy is the percentage of buggy changes for which at least one actually buggy line is ranked within the Top-k position in our returned list of sorted code lines. In terms of one buggy change i, if at least one of the Top-k most likely buggy lines returned by our approach is actually the buggy location, we consider the localization to be successful, and set the top-k value of this change $Top\_k(i)$ to 1; otherwise, we consider the localization as unsuccessful, and set the Top-k value $Top\_k(i)$ to 0. Given a set of N buggy changes in a project P, its Top-k accuracy is computed as $Top\_k(P) = \frac{1}{N}\sum_{i=1}^{N} Top\_k(i)$. In this paper, we set k = 1 and 5.

3.3 Research Questions

We formalize our study using the following three research questions:

RQ1: Can we effectively locate the buggy lines when identifying a buggy change using our proposed framework? In this RQ, we evaluate the effectiveness of our proposed framework for JIT defect identification and localization. To answer this RQ, we conduct an empirical study to evaluate our framework on 14 open source projects. Additionally, we compare its performance with two baselines, i.e., a random guessing and a static bug finder baseline (i.e., PMD [39]).

RQ2: What is the impact of using prior buggy code for JIT defect localization phase? By default, we build our model for JIT defect localization by training on prior clean code (i.e., a clean language model). In this RQ, we investigate what if we build the source code language model using buggy code (i.e., a buggy language model).

RQ3: How effective is our framework in a cross-project setting? In this RQ, we aim to explore the effectiveness of our framework in a cross-project modeling setting. By default, we build our model for each project by learning from the historical code changes within that project. To answer this RQ, we build a cross-project model by learning from the other studied projects except one project (the testing project) at a time.

Validation setting for answering the above RQs. In order to simulate the practical usage of our framework, we adopt a time-aware validation setting that divides the training and testing sets as done by prior studies [12], [25]. In our time-aware validation, for each project, we first rank all changes in chronological order according to the commit date and time. Then we use the early 60% of the changes as our training set, and use the remaining 40% of the changes as our testing set.

For answering RQ1 and RQ2, we use the within-project validation setting, i.e., learning from the early 60% changes within the project, and testing on the remaining 40% changes. For answering RQ3, we build the approach by learning from all the changes from the other studied projects except one project (the testing project) at a time. Note that in order to make a fair comparison, we keep the testing set of within-project and cross-project setting the same for answering RQ3.

4 Experimental Results

In this section, we aim to answer the aforementioned three research questions. In RQ1, we evaluate the performance of our proposed JIT defect localization approach on 14 open source projects and compare it with two baselines. In RQ2, we present the results of clean vs. buggy model. In RQ3, we show the effectiveness of our approach when using cross-project modeling.

4.1 RQ1: Performance of proposed framework

Approach: To answer this research question, we conduct an empirical study on 14 open source projects. By default in JIT defect localization phase, we build a clean model by training on a clean corpus of lines of code from prior code changes for each project. For each change in testing set, we first classify it as buggy or clean using the built logistic classifier learning on our training set (i.e., JIT defect identification). Then, for a likely buggy change determined by previous classifier, we perform our localization phase. Additionally, in order to compare the effectiveness of the proposed approach with similar solutions, we implement two baselines, i.e., random guess and a static bug finder baseline, PMD.

TABLE 3: Descriptive statistics for our training and testing sets, as well as the performance measures for our JIT defect identification approach.

Project	#Training	#Testing	#Buggy in testing	Identification ratio	Misidentification ratio
Deeplearning4j	5,262	3,508	1,251	0.909	0.282
Jmeter	8,775	5,850	642	0.732	0.198
H2o	13,148	8,766	420	0.924	0.249
Libgdx	7,811	5,208	680	0.962	0.445
Jetty	8,882	5,922	582	0.787	0.254
Robolectric	4,251	2,834	388	0.655	0.167
Storm	5,291	3,528	250	0.976	0.252
Jitsi	7,565	5,043	455	0.635	0.159
Jenkins	14,258	9,506	586	0.935	0.253
Graylog2-server	8,221	5,481	482	0.842	0.132
Flink	7,189	4,793	747	0.973	0.684
Druid	3,250	2,167	430	0.984	0.373
Closure-compile	6,522	4,348	58	0.690	0.240
Activemq	5,923	3,948	571	0.800	0.290
Average	*7,596*	*5,064*	*539*	*0.843*	*0.284*

Baseline 1: Random Guess (RG). RG is usually adopted as a baseline when there is no previous method for addressing the same research question [40]. In random guess JIT defect localization, the model randomly sort the introduced lines. In terms of computing its performance, since the performance only relates to the order of lines, we sort the introduced lines in a buggy change randomly and we repeat the random sorting 100 times to get the median performance.

Baseline 2: PMD. PMD is a commonly-used static bug finder. We choose PMD as a baseline because it is a popular static bug finder tool and has been used in prior related studies [19], [41]. PMD produces line-level warnings and assigns a priority for each warning. Another popular static bug finder tool is FindBugs [42]. We did not choose FindBugs as a baseline because it needs to operate on the Java bytecode. Prior studies have found that not all changes leave the code base in a compilable state [43]. The main reason for a broken snapshot (i.e., cannot be compiled) is due to the problems related to the resolution of dependencies [44], [45]. Since we aim to perform the evaluation on each change, the existence of broken snapshots prevents us from adding FindBugs as another baseline.

We simply execute the existing PMD tool[3] for each code change to implement the baseline. Specifically, for each change in our test set, we checkout the changed/added files after the commit time. Subsequently, we use the PMD tool to scan the changed files then record the warning priority (1-5, denotes "High", "Medium high", "Medium", "Medium low", "Low" respectively) of each introduced line by the change. If a line is not marked by PMD, we assign the value of priority as 6 (i.e., denotes "Clean"). In this way, we can sort the lines according to their warning priority (i.e., 1-6). Additionally, since some lines might have equal priority, we add a small random amount from $[0, 1]$ to all line priority values for sorting. This step simulates the developer randomly choosing to inspect the lines returned by the PMD tool with the same priority level (as proposed by a prior study [19]). Then we sort the introduced lines according to these computed priority values (i.e., "1-6" + "[0,1]") in ascending order. The lines sorted at the top of the list are more likely to be the defect location. To reduce the bias of our randomly added amount from $[0, 1]$, we repeat this process for 100 times and get the median performance for each change.

To conduct a fair comparison, we use the same approach in terms of JIT defect identification phase for our framework and two baselines. Because our point is to compare the localization performance supposing we use the same JIT defect identification tool in the first phase as described in section 2.

To investigate whether the difference between the baseline models and our proposed models is statistically significant, we employ the Wilcoxon signed-rank test [46] with a Bonferroni correction [47] at 95% confidence level. The Wilcoxon signed-rank test is a non-parametric hypothesis test which can compare two matched samples to assess whether their population mean ranks differ. Bonferroni correction is used to counteract the problem of multiple comparisons. Additionally, we employ Cliff's delta to measure the effect size. Cliff's delta is a non-parametric effect size measure that can evaluate the amount of difference between two approaches. It defines a delta of less than 0.147, between 0.147 and 0.33, between 0.33 and 0.474 and above 0.474 as "Negligible (N)", "Small (S)", "Medium (M)", "Large (L)" effect size, respectively [48].

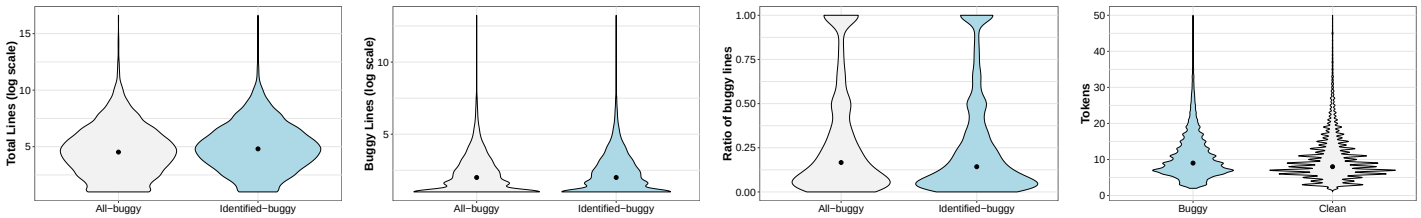
Results of JIT defect identification. Table 3 shows the result of the first phase (i.e., JIT defect identification) of our framework. For each project, we list the number of training and testing changes and two performance measures for JIT defect identification:

(1) **Identification ratio** is the recall of our JIT defect identification phase. It indicates the ratio of correctly identified buggy changes among all buggy changes in our testing set. Our JIT defect localization approach is only useful when examining a buggy change. Thus, a high identification ratio for the first phase of our framework is an essential first step for using our localization approach.

(2) **Misidentification ratio** is the false positive rate of our JIT defect identification phase. It indicates the ratio of misidentified clean changes on which we would have wasted our limited inspection effort. It is unrealistic to inspect all changes in practice. Hence, we cannot classify all changes as "buggy" to achieve an identification ratio of 1. This would lead to a large amount of wasted inspection effort. Thus, we use misidentification ratio to capture the amount of wasted effort.

From Table 3, we observe that we can achieve an average identification ratio of 0.843, and a misidentification

3. https://pmd.github.io/



(a) Total Added Lines (b) Buggy Lines (c) Buggy Lines / Total Added Lines (d) Number of tokens

Fig. 2: Distribution of total added lines, buggy lines, ratio of buggy lines relative to the added lines for all-buggy and identified-buggy changes, and the number of tokens in buggy and clean lines in our dataset. "Buggy Lines/ Total Lines" can capture the difficulty degree for a defect localization approach. A dot represents the median value.

ratio of 0.284 for JIT defect identification phase. Since the first phase is not the main contribution of our work, we simply implement a prior approach which provides us with reasonable performance already. The core focus of this paper is to investigate the performance of our proposed framework for JIT defect localization using the correctly identified buggy changes from the first phase. Additionally, we report the performance of our localization approach in a hypothetically setting which assumes that we have at hand a perfect JIT defect identification approach that is capable of identified all buggy changes and not misidentifying any changes.

Data distributions of lines and tokens. To gain an overview of the distribution of lines and tokens in our dataset, we use violin plots [49]. Since the absolute value of added lines for some changes are very large (e.g., more than 1K LOC are modified), we applied a standard log transformation (base 2) to the size of each change before visualization. Figure 2 presents the distributions of change size (i.e., added lines), buggy lines, the ratio of buggy lines relative to the number of added lines in a buggy change for identified-buggy and all-buggy changes, and the number of tokens in buggy and clean lines across all the studied projects. The ratio of buggy lines can capture the defect localization difficult degree. The smaller the ratio is, the more difficult it is to perform defect localization. A ratio of 1 indicates that all the added lines are buggy lines, the defect localization is easiest in this case. In summary, the figure shows that:

(1) Considering all-buggy changes, the majority of the buggy changes added less than 180 (i.e., $2^{7.5}$) lines of code, the median value is near 32 (i.e., 2^5) from Figure 2(a). Among the added lines, most of the buggy changes added less than 6 (i.e., near $2^{2.5}$) buggy lines from Figure 2(b). From the ratio of buggy lines, most of the buggy changes have a relatively higher difficulty degree for localization, the median value of the ratio of buggy lines is near 0.125 (i.e., locating 1 buggy line from 8 added lines) from Figure 2(c).

(2) Considering identified-buggy changes, the number of small changes (e.g., less than $2^{2.5}$) is much less compared to all-buggy changes from Figure 2(a). This means that our JIT defect identification phase tends to identify "large" buggy changes. Meanwhile, the number of buggy changes with a high difficulty degree for localization is much larger than all-buggy changes from Figure 2(c). This means that identified-buggy changes tend to be more difficult for localization.

(3) From Figure 2(c), we observe that there are a few changes that have a ratio of 1 (i.e., all the added lines are buggy lines). In this case, a JIT defect localization is not needed. However, in order to consider a more realistic usage scenario, we did not remove these changes.

(4) Figure 2(d) presents the distribution of the number of tokens of buggy and clean lines. Since a few lines may contain a large number of tokens (e.g., a long string), we only visualize the lines that contain less than 50 tokens. We use the Wilcoxon rank-sum test [50] to analyze the statistical significance of the difference between the number of tokens of buggy and clean lines. The result shows that the number of tokens of buggy lines is significantly larger than that of clean lines (i.e., $p\text{-}value < 0.05$). The number of tokens could be an important factor affecting the entropy of a line. The longer lines may have an overall average entropy higher than shorter lines. The statistical results in Figure 2(d) could be a correlated effect on why buggy lines tend to have a higher average entropy.

Results of JIT defect localization. Tables 4, 5, and 6 present the results of MRR, MAP, Top-k accuracy for each project considering identified-buggy and all-buggy changes. We list the performance of our approach and the two baselines (i.e., RG and PMD). Additionally, we list the average improvement ratio over the baseline for each project (i.e., "Improve.RG" and "Improve.PMD").

Calculation of improvement ratio. Since the performance for each project is the average performance among considered identified-buggy or all-buggy changes, there is an improvement ratio for each change (change-level improvement). The change-level improvement ratio is computed as $(\frac{Ours-Baseline}{Baseline})*100\%$ when our approach is better than baseline on this change; as $\frac{Ours-Baseline}{Ours}*100\%$ when our approach is worse than baseline on this change (now the improvement is a negative number). Then we average all the change-level improvement ratio among all considered changes for each project.

We use average improvement ratio on change-level rather than computing the improvement ratio on the final project-level. This is because there are some small changes (where most of the added lines are buggy) will flat the improvement ratio. For example, if there are two identified-buggy changes in testing set in a project, one change is with 2 added code lines where there is 1 buggy line; the

TABLE 4: The performance of MRR considering identified-buggy and all-buggy changes. "Improve" indicates the average improvement ratio over the baseline for each project. The best performance among the three approaches is highlighted in bold.

Project	Identified-buggy					All-buggy				
	RG	Improve.RG	PMD	Improve.PMD	Ours	RG	Improve.RG	PMD	Improve.PMD	Ours
Deeplearning4j	0.423	39.8%	0.459	36.4%	**0.533**	0.432	39.1%	0.467	36.3%	**0.541**
Jmeter	0.361	45.7%	0.420	21.6%	**0.443**	0.453	36.2%	0.513	13.9%	**0.519**
H2o	0.248	65.6%	0.278	16.0%	**0.326**	0.281	120.7%	0.312	106.4%	**0.357**
Libgdx	0.466	77.6%	0.484	79.1%	**0.541**	0.471	74.8%	0.488	74.4%	**0.545**
Jetty	0.338	99.4%	0.360	106.1%	**0.440**	0.418	80.0%	0.433	86.3%	**0.502**
Robolectric	0.375	75.9%	0.393	54.7%	**0.450**	0.433	52.9%	0.460	24.7%	**0.498**
Storm	0.249	81.1%	0.281	67.9%	**0.283**	0.256	85.6%	0.287	73.0%	**0.291**
Jitsi	0.234	24.9%	0.252	-17.0%	**0.275**	0.371	24.6%	0.388	-5.0%	**0.420**
Jenkins	0.457	38.1%	0.485	37.6%	**0.529**	0.478	37.3%	0.506	36.6%	**0.553**
Graylog2-server	0.277	107.2%	0.321	81.8%	**0.370**	0.355	88.7%	0.386	69.4%	**0.430**
Flink	0.225	64.1%	0.262	16.2%	**0.289**	0.228	68.2%	0.268	19.7%	**0.296**
Druid	0.311	129.2%	0.348	126.3%	**0.378**	0.317	126.9%	0.353	123.5%	**0.382**
Closure-compiler	0.187	63.2%	0.208	73.0%	**0.244**	0.307	48.9%	0.313	44.2%	**0.361**
Activemq	0.346	57.0%	0.390	47.0%	**0.436**	0.401	67.0%	0.435	49.8%	**0.482**
Average	*0.321*	*69.2%*	*0.353*	*53.3%*	***0.396***	*0.372*	*67.9%*	*0.401*	*53.8%*	***0.441***
W/T/L	*0/0/14*		*0/0/14*			*0/0/14*		*0/0/14*		
p-value	*<0.001*		*<0.001*			*<0.001*		*<0.001*		
Cliff's Delta	*0.41(M)*		*0.24(S)*			*0.44(M)*		*0.27(S)*		

TABLE 5: The performance of MAP considering identified-buggy and all-buggy changes. "Improve" indicates the average improvement ratio over the baseline for each project. The best performance among the three approaches is highlighted in bold.

Project	Identified-buggy					All-buggy				
	RG	Improve.RG	PMD	Improve.PMD	Ours	RG	Improve.RG	PMD	Improve.PMD	Ours
Deeplearning4j	0.375	43.5%	0.394	39.7%	**0.449**	0.387	42.0%	0.406	37.5%	**0.460**
Jmeter	0.340	43.0%	0.375	29.5%	**0.405**	0.435	34.6%	0.475	21.0%	**0.490**
H2o	0.232	76.8%	0.251	42.1%	**0.297**	0.266	88.2%	0.287	63.6%	**0.329**
Libgdx	0.433	78.2%	0.448	81.0%	**0.510**	0.438	76.5%	0.452	79.5%	**0.516**
Jetty	0.302	83.9%	0.320	65.3%	**0.378**	0.380	69.2%	0.398	54.0%	**0.448**
Robolectric	0.337	54.9%	0.347	43.1%	**0.389**	0.398	45.2%	0.416	31.2%	**0.455**
Storm	0.226	61.2%	0.239	66.2%	**0.261**	0.233	61.4%	0.246	66.1%	**0.266**
Jitsi	0.194	49.9%	0.203	26.6%	**0.230**	0.334	43.9%	0.347	24.9%	**0.382**
Jenkins	0.430	40.4%	0.453	35.2%	**0.500**	0.452	38.9%	0.474	34.1%	**0.523**
Graylog2-server	0.255	106.0%	0.282	84.9%	**0.324**	0.333	89.2%	0.354	72.4%	**0.391**
Flink	0.202	67.4%	0.221	42.0%	**0.254**	0.205	68.9%	0.226	43.5%	**0.259**
Druid	0.285	131.0%	0.305	133.7%	**0.344**	0.291	129.0%	0.310	132.0%	**0.349**
Closure-compiler	0.166	63.4%	0.172	36.5%	**0.211**	0.282	51.5%	0.283	22.9%	**0.334**
Activemq	0.315	57.8%	0.344	37.3%	**0.391**	0.370	63.6%	0.394	45.4%	**0.444**
Average	*0.292*	*68.4%*	*0.311*	*54.5%*	***0.353***	*0.343*	*64.4%*	*0.362*	*52.0%*	***0.403***
W/T/L	*0/0/14*		*0/0/14*			*0/0/14*		*0/0/14*		
p-value	*<0.001*		*<0.001*			*<0.001*		*<0.001*		
Cliff's Delta	*0.38(M)*		*0.27(S)*			*0.41(M)*		*0.24(S)*		

TABLE 6: The performance of Top-k accuracy considering identified-buggy and all-buggy changes.

Project	Top-1 (Identified-buggy)			Top-5 (Identified-buggy)			Top-1 (All-buggy)			Top-5 (All-buggy)		
	RG	PMD	Ours	RG	PMD	Ours	RG	PMD	Ours	RG	PMD	Ours
Deeplearning4j	0.192	0.269	**0.383**	**0.744**	0.718	0.730	0.208	0.283	**0.394**	**0.741**	0.715	0.730
Jmeter	0.157	0.247	**0.274**	0.628	0.640	**0.681**	0.257	0.352	**0.357**	0.715	0.720	**0.751**
H2o	0.119	0.144	**0.196**	0.363	0.425	**0.459**	0.152	0.181	**0.231**	0.393	0.455	**0.483**
Libgdx	0.313	0.338	**0.422**	0.661	**0.688**	0.682	0.318	0.341	**0.428**	0.665	**0.693**	0.685
Jetty	0.183	0.216	**0.312**	0.524	0.526	**0.592**	0.265	0.292	**0.376**	0.605	0.601	**0.655**
Robolectric	0.232	0.240	**0.315**	0.551	0.551	**0.598**	0.289	0.314	**0.369**	0.629	0.624	**0.647**
Storm	0.123	**0.160**	0.152	0.377	0.414	**0.422**	0.132	**0.168**	0.160	0.380	0.416	**0.428**
Jitsi	0.107	0.121	**0.166**	0.346	**0.374**	0.363	0.224	0.246	**0.301**	0.532	**0.538**	0.536
Jenkins	0.276	0.318	**0.376**	0.686	0.697	**0.717**	0.299	0.340	**0.406**	0.705	0.715	**0.734**
Graylog2-server	0.116	0.180	**0.232**	0.478	0.480	**0.532**	0.199	0.245	**0.290**	0.552	0.554	**0.598**
Flink	0.089	0.133	**0.165**	0.366	0.382	**0.425**	0.092	0.141	**0.171**	0.369	0.388	**0.435**
Druid	0.191	0.234	**0.260**	0.409	0.452	**0.508**	0.198	0.240	**0.263**	0.416	0.456	**0.512**
Closure-compiler	0.100	0.100	**0.150**	0.250	**0.300**	**0.300**	0.224	0.190	**0.276**	0.362	**0.448**	0.414
Activemq	0.175	0.232	**0.300**	0.538	0.580	**0.604**	0.231	0.277	**0.347**	0.597	0.630	**0.646**
Average	*0.170*	*0.209*	***0.265***	*0.494*	*0.516*	***0.544***	*0.221*	*0.258*	***0.312***	*0.547*	*0.568*	***0.590***
W/T/L	*0/0/14*	*1/0/13*		*1/0/13*	*2/1/11*		*0/0/14*	*1/0/13*		*1/0/13*	*3/0/11*	
p-value	*<0.001*	*<0.001*		*<0.001*	*<0.01*		*<0.001*	*<0.01*		*<0.001*	*<0.05*	
Cliff's Delta	*0.57(L)*	*0.35(M)*		*0.18(S)*	*0.12(N)*		*0.61(L)*	*0.4(M)*		*0.21(S)*	*0.11(N)*	

other change is with 200 added lines where there is 2 buggy lines. Obviously, the second change is more difficult for localization. Supposing the MRR of the two changes are 1 and 0.1 using PMD, the MRR of the two changes are 1 and 0.3 using our approach. Now the improvement ratio on project-level is: $\frac{(1+0.3)/2-(1+0.1)/2}{(1+0.1)/2} = \frac{(0.65-0.55)}{0.55} = 18\%$. However, this project-level improvement ratio cannot describe the actual improvement of our approach due to the different difficulty degree for localization on the two changes as Figure 2 shows. If we use a change-level calculation method, the average improvement ratio is: $\frac{(1-1)/1+(0.3-0.1)/0.1}{2} = \frac{(0+200\%)}{2} = 100\%$. We believe that this improvement ratio can describe the actual difference when comparing two approaches, since this is a pair-wise comparing manner. As a result, although there is not a significant difference in terms of the average performance sometimes, there could be a significant difference in terms of the average change-level improvement ratio. Additionally, note that we did not calculate the improvement ratio for Top-k accuracy, because the performance is either 1 or 0 for each change in terms of Top-k accuracy for each change.

From Tables 4, 5, and 6, we have the following observations:

(1) On average across the 14 projects, our approach achieves a reasonable performance. It achieves an MRR of 0.396, an MAP of 0.353, a top-1 accuracy of 0.265 and a top-5 accuracy of 0.544 considering identified-buggy changes. From MRR, the performance means that our approach can successfully locate the first buggy line in near 3 lines on average. From Top-k accuracy, the performance means that our approach can successfully locate at least one buggy line in top-1 line among 26.5% and in top-5 lines among 54.0% of identified-buggy changes.

(2) Among our identified-buggy changes, there are a few small buggy changes (i.e., adding $<= 5$ lines). In this case, top-5 accuracy is 1 for both our framework and baselines. However, we observe that there are only near 0.1% of small changes (i.e., adding $<= 5$ lines) in identified-buggy changes. Thus, we believe that our top-5 accuracy is reasonable.

(3) Considering all-buggy changes, the performance is better from Tables 4, 5, and 6. This is observed because the identified-buggy changes are likely larger and more difficult for localization as Figure 2 shows. The median value of added lines of identified-buggy is a bit larger than that of all-buggy changes. The ratio of buggy lines of identified-buggy is smaller than that of all-buggy changes (the ratio is smaller; the localization tends to be more difficult).

(4) Comparing to RG and PMD, our approach outperforms the two baselines in terms of all the measures on average with a statistical significance (i.e., p-value < 0.05) and a non-negligible effect size according to Wilcoxon signed-rank test and Cliff's delta in most of the projects from Tables 4, 5, and 6. Additionally, the improvement ratio is substantial on average (e.g., at least greater than 50% in terms of MAP and MRR). The row "W/T/L" reports the number of projects for which the corresponding approach obtains a better, equal or worse performance than ours. Our framework outperforms the two baselines in all of the projects in terms of MRR,and MAP, and in most of the projects in terms of Top-k accuracy.

(5) In project "Jitsi", although the performance of ours is better, the improvement ratio is a negative value in terms of MRR compared to PMD. Such value indicates that our framework may be failed for some changes, but the PMD is successful for defect localization for these changes and with a significant improvement ratio compared to our framework. In terms of Top-k accuracy, although the PMD baseline could perform better than our framework in a few projects (e.g., Libgdx, Storm, and Jitsi), our framework outperforms PMD with a statistically significant difference among 14 projects.

(6) Note that some of changes are with a high ratio of buggy lines relative to the added lines as Figure 2(c) shows; for those changes, most of (or even all of) the added lines are buggy. These changes would lead to a good performance in terms of all the measures and across all approaches. Such changes are likely to be the reason for the reasonably good performance of the random guess baseline.

Our approach can achieve a reasonable localization performance that can outperform the two baselines with a statistical significance in average of the 14 projects.

4.2 RQ2: Effectiveness of buggy model

Approach: By default, we build our JIT defect localization model by training on the previously added clean source code lines (i.e., a clean language model). The assumption of a clean model is that buggy code tends to be more surprising with clean code. In this RQ, we investigate how effective if we build our model using the buggy source code lines (i.e., a buggy model). The assumption of a buggy model is that buggy code tends to be similar with prior buggy code.

For building a buggy model, we use all the added buggy lines of the training set as the training corpus for building a language model. However, the method for computing the line entropy and sorting is different with the default clean model, since the assumption is different. In the buggy model, a lower entropy value means a higher likelihood to be defective. Thus, we sort the lines in ascending order according to their entropy value.

In detail, in buggy model, we compute the line entropy (denoted as $H_p(s)_b$) by subtracting the average entropy from the minimum (i.e., min) entropy of its tokens as Formula 5 shows. The reasons for this adjustment are two-folds:

1) The min entropy captures the most natural token sequences of a line. Within the context of the buggy model, the most natural line means the buggiest line.

2) The average entropy captures the entire naturalness of a line, subtracting it from the min entropy is useful for describing the entire naturalness of the line, especially when the min entropy of different lines might be equal.

$$H_p(s)_b = min(H_p(t_1), ..., H_p(t_{|s|})) - \frac{1}{|s|}\sum_{i=1}^{|s|} H_p(t_i) \quad (5)$$

Results: We evaluate the buggy model on the same dataset and setting used for answering RQ1. Table 7 presents the performance of buggy model and its comparison with the default clean model considering identified-buggy changes.

TABLE 7: The performance buggy vs. clean model considering identified-buggy changes.

Project	MRR		MAP		Top-1		Top-5	
	Buggy	Clean	Buggy	Clean	Buggy	Clean	Buggy	Clean
Deeplearning4j	0.483	**0.533**	0.422	**0.449**	0.325	**0.383**	0.682	**0.730**
Jmeter	0.422	**0.443**	0.384	**0.405**	**0.279**	0.274	0.587	**0.681**
H2o	0.305	**0.326**	0.286	**0.297**	0.188	**0.196**	0.415	**0.459**
Libgdx	0.523	**0.541**	0.500	**0.510**	0.401	**0.422**	0.665	**0.682**
Jetty	0.391	**0.440**	0.351	**0.378**	0.255	**0.312**	0.563	**0.592**
Robolectric	0.443	**0.450**	**0.391**	0.389	0.307	**0.315**	**0.602**	0.598
Storm	**0.326**	0.283	**0.272**	0.261	**0.201**	0.152	**0.471**	0.422
Jitsi	0.212	**0.275**	0.194	**0.230**	0.100	**0.166**	0.304	**0.363**
Jenkins	0.494	**0.529**	0.474	**0.500**	0.339	**0.376**	0.712	**0.717**
Graylog2-server	0.358	**0.370**	0.318	**0.324**	0.219	**0.232**	0.515	**0.532**
Flink	0.266	**0.289**	0.240	**0.254**	0.140	**0.165**	0.418	**0.425**
Druid	0.352	**0.378**	0.322	**0.344**	0.253	**0.260**	0.442	**0.508**
Closure-compiler	0.211	**0.244**	0.201	**0.211**	**0.150**	**0.150**	0.200	**0.300**
Activemq	0.430	**0.436**	0.385	**0.391**	0.298	**0.300**	0.584	**0.604**
Average	*0.373*	***0.396***	*0.339*	***0.353***	*0.247*	***0.265***	*0.512*	***0.544***
p-value	*<0.01*		*<0.01*		*<0.05*		*<0.01*	
Cliff's Delta	*0.14(N)*		*0.11(N)*		*0.12(N)*		*0.15(S)*	
W/T/L	*1/0/13*		*2/0/12*		*2/1/11*		*2/0/12*	

Since identified-buggy changes are more realistic than all-buggy for using our approach, we only consider identified-buggy changes in the following sections. From the table, we observe that:

(1) Comparing the performance of buggy model to two baselines in RQ1 as shown in Tables 4, 5, and 6, we observe that buggy model can achieve a comparable or better performance in most of the cases on average. This indicates that a language model based on buggy code can also be possible for JIT defect localization.

(2) The clean model is better than the buggy model on average in terms of all the measures considering identified-buggy changes. Clean model outperforms buggy model with a statistical significance (i.e., $p - value < 0.05$) and a non-negligible effect size according to Wilcoxon signed-rank test and Cliff's delta in terms of MAP and top-5 accuracy. There is no statistical difference between them in terms of MRR and top-1 accuracy.

(4) The main reason for the performance difference between the clean model and the buggy model might be the assumption behind them. The assumption of the clean model is that buggy code tends to be surprising compared to the prior clean code. For example, our approach successfully located the buggy line at the first position in the change *"a7747417e541666d5316bccaaf41ca5e1112e985"* for the Deeplearining4j project, since the buggy line contains *"return feedForward(true, excludeOutputLayers, false, true);"*. We noticed that the use of *"return feedForward(true"* rarely occurred elsewhere in the training corpus. After a closer investigation, we observed that the most commonly used token after *"new feedForward("* is *"train"*, or *")"* in the training corpus. Thus, this line has a rather high entropy since it is rather surprising relative to the training corpus. The corresponding bug-fix change fixed this line by using *"return feedForward(train, excludeOutputLayers, false, true);"*. The assumption of the buggy model is that buggy code tends to be similar compared to prior buggy code. For example, the buggy model tends to sort the lines that contain the use of *"new INDArrayIndex[]"* at the top of the list in the Deeplearining4j project (e.g., in change *"33c68cdf23257cc9f09d721625f47cd832de8ca6"*) since the *"new INDArrayIndex[]"* occurs many times in the buggy training corpus. However, in this change, the line used the *"new INDArrayIndex[]"* is not the buggy line. Thus, the buggy model failed to locate this buggy change.

(5) In most cases, the clean model outperforms the buggy model. Hence, supporting our assumption that the clean model is more effective for JIT defect localization. However, the clean model does not consistently outperform the buggy model. This indicates that the assumption of the buggy model might be also appropriate in other cases.

A buggy model can also be possible for JIT defect localization. However, a clean model is a better choice on average.

4.3 RQ3: Performance of cross-project model

Approach: For each target project, we build our approach by learning from all the other projects in our study. Similar to RQ2, we focus on the clean model since we observed that the clean model is better than the buggy model. And we focus on identified-buggy changes, since they are more realistic for using our approach.

In our cross-project setting, we build the N-gram model by learning from all the clean corpus of other projects. To do so, we first combine all the clean source lines of other projects as one multi-project training corpus. Second, we build the N-gram model by learning on the combined multi-project corpus using the same modeling setting as used for answering RQ1. Third, we apply the multi-project model on the testing set which consists of a single target project. We then compare the performance of our cross-project setting with our default within-project setting.

Results: Table 8 presents the effectiveness of cross-project modeling. The results show that:

(1) On average across the 14 projects, the cross-project model achieves a comparable performance compared to within-project model. There is no statistical significance between them.

(2) We can observe that cross-project model can slightly improves the performance on average. From the above observation, we can conclude that cross-project is also doable. The reason might be that the training corpus in cross-project is much larger than the corpus for the within-project setting. For example, when targeting the Deeplearning4j project, the total tokens in the training corpus is 3,110K and 40,620K (above 10 times larger) in within-project and cross-project setting respectively. In other words, the cross-project

TABLE 8: The performance cross-project (CP) vs. within-project (WP) model considering identified-buggy changes.

Project	MRR		MAP		Top-1		Top-5	
	WP	CP	WP	CP	WP	CP	WP	CP
Deeplearning4j	0.533	**0.571**	0.449	**0.475**	0.383	**0.427**	0.730	**0.752**
Jmeter	0.443	**0.460**	0.405	**0.417**	0.274	**0.311**	**0.681**	0.666
H2o	0.326	**0.336**	0.297	**0.308**	0.196	**0.206**	0.459	**0.474**
Libgdx	**0.541**	0.516	**0.510**	0.488	**0.422**	0.394	**0.682**	0.671
Jetty	**0.440**	0.425	**0.378**	0.373	**0.312**	0.282	**0.592**	**0.592**
Robolectric	0.450	**0.471**	0.389	**0.404**	0.315	**0.339**	0.598	**0.630**
Storm	0.283	**0.315**	0.261	**0.277**	0.152	**0.201**	0.422	**0.447**
Jitsi	**0.275**	0.236	**0.230**	0.209	**0.166**	0.135	**0.363**	0.315
Jenkins	**0.529**	0.523	**0.500**	0.498	**0.376**	0.363	0.717	**0.732**
Graylog2-server	0.370	**0.383**	0.324	**0.333**	0.232	**0.244**	**0.532**	0.530
Flink	**0.289**	0.285	**0.254**	0.250	**0.165**	0.162	**0.425**	0.415
Druid	0.378	**0.385**	0.344	**0.347**	0.260	**0.284**	**0.508**	0.478
Closure-compiler	0.244	**0.271**	**0.211**	0.195	0.150	**0.175**	0.300	**0.325**
Activemq	0.436	**0.445**	0.391	**0.398**	**0.300**	0.293	0.604	**0.628**
Average	*0.396*	***0.401***	*0.353*	***0.355***	*0.265*	***0.273***	*0.544*	***0.547***
W/T/L	*5/0/9*		*6/0/8*		*6/0/8*		*6/1/7*	
p-value	*>0.05*		*>0.05*		*>0.05*		*>0.05*	
Cliff's Delta	*0.04(N)*		*0(N)*		*0.06(N)*		*0.02(N)*	

model achieves comparable or better performance by using approximately a 10 times larger training corpus than the within-project model.

Cross-project modelling can achieve comparable performance compared to within-project model by learning on a much larger training corpus.

5 Discussion

5.1 Effectiveness of fusing the clean and buggy models

In this subsection, we explore whether we can improve the performance of our framework by fusing the clean and buggy models. We investigate two fusing methods, i.e., fusing by entropy value (F-value) and fusing by rank (F-rank).

Fusing by entropy value (F-value). In this fusing method, we simply update the entropy for each line by subtracting the entropy of the buggy model from the entropy of the clean model. In detail, we compute the fused line entropy (denoted as H_{F_value} as Formula 6 shows. The reason for this calculation method is due to the difference between the clean and buggy models. In the clean model, a higher entropy means a higher defectiveness likelihood, we sort the lines in descending order according to their entropy value. In the buggy model, a lower entropy means a higher defectiveness likelihood, we sort the lines in ascending order according to their entropy value. In this fusing method, we use the subtraction method. As a result, the sorting method is the same as the one used in the clean model (i.e., in descending order according to H_{F_value}).

$$F\_value = H_{Clean} - H_{Buggy} \tag{6}$$

Fusing by rank (F-rank). Additionally, we explore another fusing method using the entropy rank rather than the entropy value of each line. The fused rank (F-rank) is calculated as Formula 7 shows. Then, we sort the lines by the fused entropy rank. For example, suppose a change introduced three lines of code (Line A, B, and C), the entropy rank (in ascending order) for each line in the clean model is $\{1, 2, 3\}$ respectively (i.e., Line C is the most defective line in the clean model). The entropy rank (in ascending order) of entropy for each line in a buggy model is $\{3, 2, 1\}$ respectively (i.e., Line C is also the most defective one in buggy model). The fused rank for each line is $\{-2, 0, 2\}$ respectively. Then, we sort the lines in descending order according to the fused rank. As a result, Line C is also the most defective one in this fusing method.

$$F\_rank = Rank_{Clean} - Rank_{Buggy} \tag{7}$$

Table 9 presents the results for the F-value and F-rank models, and their differences compared to the clean model. From the table, we observe that the clean model outperforms F-value and F-rank on average. There is a statistically significant difference ($p - value < 0.05$) between the clean model and the fused models in terms of MRR, MAP and Top-5 accuracy. There is no statistically significant difference ($p - value > 0.05$) in terms of Top-1 accuracy.

In summary, fusing the buggy and clean models cannot improve the performance compared to the clean model on average.

5.2 Impact of different methods for using entropy

In localization step, we sort the added lines according to the entropy of each line. By default, we calculate the line entropy by combining the average entropy and the max entropy among all the tokens of a line following Formula 4 for the clean model (i.e., Avg+Max). In the buggy model, we calculate the line entropy by combining the average entropy and min entropy among all the tokens of a line following Formula 5 (i.e., Min-Avg). In this subsection, we investigate the effectiveness of three other methods for using the line entropy.

Average (Avg). We use the average entropy of all the tokens in a line to represent the line entropy. This method can be used in both the clean and buggy models. Additionally, we use max/min as a tie-breaker for sorting when the average entropy of two lines are equal.

Max/Min. We use the max entropy of all the tokens in a line to represent the line entropy. The max entropy is used in the clean model. We use the min entropy of all the tokens in a line to represent the line entropy. The min entropy is used in the buggy model. Additionally, we use the average entropy

TABLE 9: The performance of fusing the buggy and clean models. F-value indicates the fusing method using the entropy value of the buggy and clean models. F-rank indicates the fusing method using the entropy rank of the buggy and clean models.

Project	MRR			MAP			Top-1			Top-5		
	F-value	F-rank	Clean	F-value	F-rank	Clean	F-value	F-rank	Clean	F-value	F-rank	Clean
Deeplearning4j	0.482	0.478	**0.533**	0.416	0.416	**0.449**	0.332	0.326	**0.383**	0.675	0.675	**0.730**
Jmeter	0.410	0.413	**0.443**	0.383	0.386	**0.405**	0.255	0.257	**0.274**	0.583	0.581	**0.681**
H2o	**0.329**	0.326	0.326	0.289	0.287	**0.297**	**0.216**	0.214	0.196	0.441	0.436	**0.459**
Libgdx	0.490	0.495	**0.541**	0.470	0.472	**0.510**	0.373	0.376	**0.422**	0.635	0.644	**0.682**
Jetty	0.382	0.381	**0.440**	0.342	0.342	**0.378**	0.251	0.251	**0.312**	0.515	0.513	**0.592**
Robolectric	0.435	0.433	**0.450**	0.374	0.373	**0.389**	**0.315**	0.311	**0.315**	**0.598**	0.594	**0.598**
Storm	0.301	**0.307**	0.283	0.256	0.258	**0.261**	**0.176**	**0.176**	0.152	0.426	**0.434**	0.422
Jitsi	0.281	**0.282**	0.275	**0.236**	0.231	0.230	**0.170**	**0.170**	0.166	0.377	**0.394**	0.363
Jenkins	0.491	0.492	**0.529**	0.473	0.473	**0.500**	0.350	0.350	**0.376**	0.655	0.662	**0.717**
Graylog2-server	0.355	0.351	**0.370**	0.314	0.313	**0.324**	0.222	0.217	**0.232**	0.512	0.512	**0.532**
Flink	0.296	**0.298**	0.289	0.244	0.245	**0.254**	**0.179**	0.183	0.165	0.407	0.404	**0.425**
Druid	0.353	0.355	**0.378**	0.316	0.317	**0.344**	0.262	**0.265**	0.260	0.437	0.440	**0.508**
Closure-compiler	0.239	0.223	**0.244**	0.186	0.181	**0.211**	0.100	0.100	**0.150**	**0.400**	0.350	0.300
Activemq	0.407	0.411	**0.436**	0.364	0.369	**0.391**	0.278	0.284	**0.300**	0.549	0.551	**0.604**
Average	*0.375*	*0.375*	***0.396***	*0.333*	*0.333*	***0.353***	*0.249*	*0.249*	***0.265***	*0.515*	*0.514*	***0.544***
p-value	*<0.05*	*<0.05*		*<0.01*	*<0.01*		*>0.05*	*>0.05*		*<0.05*	*<0.05*	
Cliff's Delta	*0.12(N)*	*0.12(N)*		*0.14(N)*	*0.14(N)*		*0.06(N)*	*0.06(N)*		*0.16(S)*	*0.17(S)*	

TABLE 10: The performance of different methods for calculating the line entropy in the clean model.

Project	MRR			MAP			Top-1			Top-5		
	Avg	Max	Avg+Max	Avg	Max	Avg+Max	Avg	Max	Avg+Max	Avg	Max	Avg+Max
Deeplearning4j	0.468	**0.543**	0.533	0.416	**0.466**	0.449	0.313	**0.388**	0.383	0.674	**0.748**	0.730
Jmeter	0.436	**0.488**	0.443	0.399	**0.437**	0.405	0.287	**0.332**	0.274	0.600	**0.689**	0.681
H2o	**0.326**	0.317	**0.326**	**0.297**	0.292	**0.297**	0.204	**0.198**	0.196	0.454	0.441	**0.459**
Libgdx	0.525	**0.546**	0.541	0.503	0.508	**0.510**	0.410	**0.431**	0.422	0.662	0.673	**0.682**
Jetty	0.405	0.410	**0.440**	0.363	0.367	**0.378**	0.277	0.271	**0.312**	0.544	0.566	**0.592**
Robolectric	0.424	**0.472**	0.450	0.379	**0.404**	0.389	0.287	**0.354**	0.315	0.579	0.594	**0.598**
Storm	**0.314**	0.289	0.283	**0.272**	0.258	0.261	**0.184**	**0.184**	0.152	0.467	0.369	**0.422**
Jitsi	**0.298**	0.285	0.275	0.241	**0.235**	0.230	0.176	**0.183**	0.166	0.415	0.363	**0.363**
Jenkins	0.525	**0.534**	0.529	0.489	**0.505**	0.500	0.374	**0.381**	0.376	0.724	**0.726**	0.717
Graylog2-server	0.341	0.353	**0.370**	0.312	0.318	**0.324**	0.200	0.224	**0.232**	0.498	0.495	**0.532**
Flink	0.281	**0.293**	0.289	0.246	**0.254**	**0.254**	0.155	**0.184**	0.165	0.415	0.407	**0.425**
Druid	0.374	0.377	**0.378**	**0.347**	0.346	0.344	0.267	**0.272**	0.260	0.487	0.475	**0.508**
Closure-compiler	0.258	**0.293**	0.244	0.218	**0.230**	0.211	0.150	**0.225**	0.150	0.325	**0.350**	0.300
Activemq	**0.439**	0.427	0.436	0.393	**0.402**	0.391	**0.313**	0.282	0.300	0.584	**0.604**	**0.604**
Average	*0.387*	***0.402***	*0.396*	*0.348*	***0.359***	*0.353*	*0.257*	***0.279***	*0.265*	*0.531*	*0.536*	***0.544***
p-value	*>0.05*	*>0.05*		*>0.05*	*>0.05*		*>0.05*	*>0.05*		*>0.05*	*>0.05*	
Cliff's Delta	*0.09(N)*	*0.08(N)*		*0.02(N)*	*0.04(N)*		*0.03(N)*	*0.12(N)*		*0.11(N)*	*0.04(N)*	

TABLE 11: The performance of different methods for calculating the line entropy in the buggy model.

Project	MRR			MAP			Top-1			Top-5		
	Avg	Min	Min-Avg	Avg	Min	Min-Avg	Avg	Min	Min-Avg	Avg	Min	Min-Avg
Deeplearning4j	0.443	0.406	**0.483**	0.403	0.383	**0.422**	0.286	0.249	**0.325**	0.643	0.577	**0.682**
Jmeter	0.411	0.379	**0.422**	0.376	0.364	**0.384**	0.268	0.240	**0.279**	0.581	0.532	**0.587**
H2o	0.291	0.271	**0.305**	0.274	0.258	**0.286**	0.183	0.160	**0.188**	0.397	0.358	**0.415**
Libgdx	0.519	0.493	**0.523**	0.497	0.475	**0.500**	**0.404**	0.382	0.401	0.651	0.610	**0.665**
Jetty	0.379	0.367	**0.391**	0.344	0.339	**0.351**	0.245	0.249	**0.255**	0.533	0.491	**0.563**
Robolectric	0.430	0.382	**0.443**	0.385	0.366	**0.391**	0.295	0.264	**0.307**	0.575	0.520	**0.602**
Storm	0.323	0.278	**0.326**	0.269	0.256	**0.272**	**0.205**	0.168	0.201	0.467	0.381	**0.471**
Jitsi	0.239	**0.272**	0.212	0.202	**0.229**	0.194	0.135	**0.152**	0.100	0.343	**0.388**	0.304
Jenkins	0.488	0.464	**0.494**	0.469	0.450	**0.474**	0.332	0.305	**0.339**	0.697	0.670	**0.712**
Graylog2-server	0.334	0.285	**0.358**	0.303	0.272	**0.318**	0.190	0.163	**0.219**	0.502	0.429	**0.515**
Flink	0.263	0.250	**0.266**	0.237	0.231	**0.240**	0.140	0.135	**0.140**	0.398	0.367	**0.418**
Druid	0.348	0.341	**0.352**	0.321	0.315	**0.322**	0.246	0.241	**0.253**	0.423	0.407	**0.442**
Closure-compiler	0.212	**0.234**	0.211	0.200	0.199	**0.201**	**0.150**	**0.150**	**0.150**	0.225	**0.300**	0.200
Activemq	0.422	0.410	**0.430**	0.378	0.361	**0.385**	0.287	0.282	**0.298**	0.575	0.565	**0.584**
Average	*0.365*	*0.345*	***0.373***	*0.333*	*0.321*	***0.339***	*0.240*	*0.224*	***0.247***	*0.501*	*0.471*	***0.512***
p-value	*<0.05*	*<0.05*		*<0.01*	*<0.01*		*>0.05*	*<0.05*		*>0.05*	*>0.05*	
Cliff's Delta	*0.09(N)*	*0.17(S)*		*0.07(N)*	*0.14(N)*		*0.08(N)*	*0.19(S)*		*0.1(N)*	*0.24(S)*	

TABLE 12: The performance of the likelihood method based on a classifier that is trained from the avg, min and max entropy values in the clean model.

Project	MRR		MAP		Top-1		Top-5	
	Likelihood	Avg+Max	Likelihood	Avg+Max	Likelihood	Avg+Max	Likelihood	Avg+Max
Deeplearning4j	0.439	**0.533**	0.412	**0.449**	0.277	**0.383**	0.642	**0.730**
Jmeter	0.410	**0.443**	0.387	**0.405**	0.264	**0.274**	0.591	**0.681**
H2o	0.320	**0.326**	0.296	**0.297**	**0.196**	**0.196**	0.438	**0.459**
Libgdx	0.517	**0.541**	0.499	**0.510**	0.394	**0.422**	0.653	**0.682**
Jetty	0.400	**0.440**	0.360	**0.378**	0.273	**0.312**	0.546	**0.592**
Robolectric	0.415	**0.450**	0.375	**0.389**	0.287	**0.315**	0.539	**0.598**
Storm	0.263	**0.283**	0.247	**0.261**	0.139	**0.152**	0.389	**0.422**
Jitsi	**0.298**	0.275	**0.246**	0.230	**0.183**	0.166	**0.405**	0.363
Jenkins	**0.529**	**0.529**	0.498	**0.500**	0.372	**0.376**	**0.732**	0.717
Graylog2-server	**0.385**	0.370	**0.331**	0.324	**0.246**	0.232	**0.537**	0.532
Flink	0.277	**0.289**	0.250	**0.254**	0.160	**0.165**	0.393	**0.425**
Druid	0.368	**0.378**	**0.346**	0.344	0.255	**0.260**	0.475	**0.508**
Closure-compiler	**0.262**	0.244	**0.216**	0.211	**0.150**	**0.150**	0.325	**0.300**
Activemq	**0.438**	0.436	0.390	**0.391**	**0.319**	0.300	0.573	**0.604**
Average	*0.380*	***0.396***	*0.347*	***0.353***	*0.251*	***0.265***	*0.517*	***0.544***
p-value	*<0.05*		*>0.05*		*>0.05*		*<0.05*	
Cliff's Delta	*0.15(S)*		*0.08(N)*		*0.10(N)*		*0.13(N)*	

TABLE 13: The performance of the likelihood method based on a classifier that is trained from the avg, min and max entropy values in the buggy model.

Project	MRR		MAP		Top-1		Top-5	
	Likelihood	Min-Avg	Likelihood	Min-Avg	Likelihood	Min-Avg	Likelihood	Min-Avg
Deeplearning4j	**0.496**	0.483	**0.435**	0.422	**0.353**	0.325	**0.687**	0.682
Jmeter	**0.531**	0.422	**0.450**	0.384	**0.406**	0.279	**0.683**	0.587
H2o	0.296	**0.305**	0.275	**0.286**	0.180	**0.188**	0.405	**0.415**
Libgdx	0.485	**0.523**	0.471	**0.500**	0.365	**0.401**	0.624	**0.665**
Jetty	0.362	**0.391**	0.333	**0.351**	0.227	**0.255**	0.504	**0.563**
Robolectric	0.396	**0.443**	0.361	**0.391**	0.272	**0.307**	0.524	**0.602**
Storm	0.293	**0.326**	0.256	**0.272**	0.172	**0.201**	0.414	**0.471**
Jitsi	**0.299**	0.212	**0.251**	0.194	**0.176**	0.100	**0.419**	0.304
Jenkins	**0.514**	0.494	**0.496**	0.474	**0.378**	0.339	0.673	**0.712**
Graylog2-server	**0.358**	**0.358**	0.316	**0.318**	**0.229**	0.219	0.493	**0.515**
Flink	**0.267**	0.266	0.235	**0.240**	**0.144**	0.140	0.388	**0.418**
Druid	**0.354**	0.352	**0.323**	0.322	0.251	**0.253**	**0.447**	0.442
Closure-compiler	**0.253**	0.211	**0.235**	0.201	**0.150**	**0.150**	**0.375**	0.200
Activemq	0.390	**0.430**	0.355	**0.385**	0.249	**0.298**	0.569	**0.584**
Average	***0.378***	*0.373*	***0.342***	*0.339*	***0.254***	*0.247*	***0.515***	*0.512*
p-value	*>0.05*		*>0.05*		*>0.05*		*>0.05*	
Cliff's Delta	*0.03(N)*		*0.00(N)*		*0.01(N)*		*0.05(N)*	

as a tie-breaker for sorting when the max/min entropy of two lines are equal.

Likelihood. We compute a likelihood value based on a classifier learned from avg, max, and min entropy values of a line. In this method, we calculate the avg, min and max entropy values of each line in both the training and testing set by using the built clean/buggy model. Using the avg, min and max entropy values, we build a classifier (i.e., Logistic Regression) learned from the training set. Then, the likelihood of a line in the testing set (with avg, min and max entropy values) being buggy is predicted using such a classifier. Finally, we sort in descending order the lines in each change that appears in the testing set.

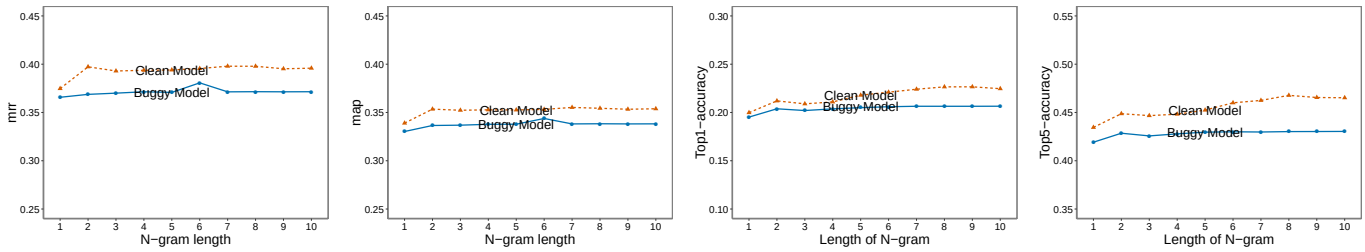
Tables 10 and 11 present the results of the different line entropy calculation methods that we considered, and the best performing one is highlighted in bold for each project and each performance measure. From these two tables, we observe that: (1) In the clean model, considering the average scores across the projects, the Max method can achieve comparable or better performance than the other methods. However, there is no statistically significant difference in the effectiveness scores of Max, Avg and Max+Avg (*p-value*>0.05). (2) In the buggy model, the Min-Avg method outperforms the Avg and Min methods for most cases. In terms of MRR and MAP, the Min-Avg method outperforms the Avg and Min methods in a statistically significant way (*p-value*<0.05).

Tables 12 and 13 present the performance of the likelihood methods in the clean and buggy models. From these two tables, we observe that: (1) In the clean model, the Avg+Max method outperforms the likelihood method on average and in a statistically significant way in terms of MRR and Top-5 accuracy. There is no statistically significant difference observed in terms of MAP and Top-1 accuracy. (2) In the buggy model, the likelihood method can achieve comparable or better performance compared to the Min-Avg method. The likelihood method outperforms Min-Avg method on average, but with no statistically significant difference.

In summary, a pure Max method using average only as the tie-breaker can achieve a comparable performance compared to our originally used method in the clean model. The likelihood (predicted by using a classifier based on min, avg and max entropy values) method can achieve a comparable performance compared to our originally used method in the buggy model.

5.3 The impact of using different configurations of N-gram

The different N-gram configurations might affect the effectiveness of modeling [24], [51]. In the prior RQs, we used the default configuration as suggested by Hellendoorn and



(a) MRR (b) MAP (c) Top1-accuracy (d) Top5-accuracy

Fig. 3: The impact of the N-gram length on the clean and buggy models.

TABLE 14: The impact of using different smoothing methods on the clean and buggy models.

Measure	Clean model				Buggy model			
	JM	AD	ADM	WB	JM	AD	ADM	WB
MRR	0.396	**0.402**	0.398	0.400	**0.381**	0.370	0.369	0.369
MAP	0.353	**0.358**	0.354	0.356	**0.344**	0.338	0.337	0.337
Top1	0.265	0.268	0.266	**0.269**	**0.255**	0.241	0.240	0.240
Top5	0.544	**0.555**	0.541	0.544	**0.516**	0.512	0.508	0.509

Devanbu [24]. In this subsection, we investigate the impact of different N-gram configurations, i.e., smoothing method and N-gram length.

Smoothing is a technique essential in the construction of n-gram models. Smoothing is used to make distributions more uniform, by adjusting low probabilities such as zero probabilities upward, and high probabilities downward. Not only does smoothing generally prevent zero probabilities, but it also improves the accuracy of the model as a whole [24], [31]. N-gram length refers to the length of the considered token sequences. N-gram models assume that each token depends on the previous $N-1$ tokens. Bigger length may be more specific, but correspondingly less frequent in the corpus, smaller length occur more often, but may lose information. Thus, we are interested in how these configurations impact the performance of our model and which choice is better. We consider four popular smoothing methods. We briefly introduce their ideas below. For more detailed description of these smoothing methods please refer to the work by Chen and Goodman [31].

Smoothing methods:

Jelinek-Mercer (JM). JM involves a linear interpolation of the maximum likelihood model with the collection model, using a coefficient to control the amount of smoothing. It is a linear interpolation method for combining the information from lower-order n-gram models for estimating higher-order probabilities.

Witten-Bell (WB). WB smoothing is developed for the task of text compression and is considered as an instance of the Jelinek-Mercer smoothing. WB also uses linear discounting but differs in that it incorporates the interpolation coefficient based on the average frequency of events seen in a context.

Absolute Discounting (AD). AD method lowers the probability of seen words by subtracting a constant from their counts. Different from the Jelinek-Mercer method, AD discounts the seen word probability by subtracting a constant instead of multiplying it by a coefficient.

Absolute Discounting Modified (ADM). ADM is a modified AD. ADM improves AD by using three separate discounts, for events seen once, twice and more than twice [24].

N-gram length. In terms of the N-gram length, we built N-gram models with lengths from one (i.e., unigram) to ten in order to investigate the impact of the N-gram length on the model performance. We conducted the experiments in this discussion using the same dataset, keeping the same validation setting and smoothing method as used in RQ1.

Table 14 presents the results of different smoothing methods. In terms of both the buggy and clean models, we list the average performance of all projects by using different smoothing methods. From this table, we observe that the smoothing method has no substantial impact on the average performance. In general, for the buggy model, the best choice is Jelinek-Mercer (JM) smoothing. For the clean model, the best choice is Absolute Discounting (AD) smoothing. Figure 3 presents the average performance of all projects across different N-gram length. We can observe that the N-gram length has no substantial impact on the average performance. In general, bigger lengths (e.g., 8-gram) tend to perform better.

In summary, the n-gram configuration has no substantial impact on the average performance. In general, the Jelinek-Mercer (JM) smoothing tends to be better for the buggy model. The Absolute Discounting (AD) smoothing tends to be better for the clean model. A bigger N-gram length (e.g., 8-gram) tends to perform better for both the clean and buggy models.

5.4 Implications

Usage scenarios, benefits and costs of using our tool. Our tool consists of a JIT defect identification phase and a JIT defect localization phase. The typical usage scenario of our tool is to provide suggestions on suspicious buggy code lines introduced by a buggy change at check-in time. For example, Bob is a developer in a large project team, and his main responsibility is to inspect code changes that are submitted by other developers. He is typically assigned

to inspect more than 50 code changes in a single day. By default, we can suppose Bob used the current JIT defect identification tool as described in Section 2.

Without our tool. Following the recommendations of the used JIT defect identification tool, Bob needs to inspect 20 likely buggy changes that are identified as buggy in a day. Suppose one change introduced 180 lines on average, he needs to inspect $20 \cdot 180 = 3,600$ lines of code. He finds that it is hard for him to focus when he reviews more than 1,000 lines of code, and he often introduces errors when inspecting the remaining lines.

With our tool. Bob uses the same JIT defect identification tool (i.e., the first phase is the same). Thus, Bob also needs to inspect 20 likely buggy changes. With our tool, he may only need to inspect a small list of lines for each of these 20 changes (e.g., top-5 or top-10 lines as suggested by our tool). As a result, he can pay more attention to inspect the most suspicious lines of code in each change. In this way, he can spend less time and effort to locate the exact bug positions.

The benefits is to save the inspection time and effort, especially when there is limited time and source for inspecting code changes. The cost is that we may miss a few bugs and waste the effort on false positives. Since our approach is the following step after JIT defect identification, the cost of our approach is associated with the performance of the JIT defect identification step. Hence the cost of our approach includes that: (1) we missed 15.7% of the buggy changes, since these truly buggy changes are not identified correctly in the first step; (2) we wasted inspection effort on 28.4% of the clean changes, since they are false positives. Additionally, developers cannot simply analyze the warned buggy lines in isolation, but rather, they need the surrounding context lines to fully understand them.

Implications for practitioners. First, we found that in most of the buggy changes, the ratio of buggy lines among all the added lines is smaller than 0.25; half of them is near 0.125. This data highlights the necessary for JIT defect localization, since it is impractical to inspect all the changed lines for each change with limited inspection effort. Second, using our approach, possibly buggy lines can be highlighted in an early stage (i.e., at check-in time), developers can inspect them early when they are still fresh with the context. The benefit is that developers can locate the defect by only inspecting a list of highly defective lines suggested by our approach. Third, there are two ways for building a localization model, i.e., a clean model and a buggy model. Clean model might be better for most of the projects, since buggy code tends to be surprising compared to clean code. Fourth, when applying our approach on a new project, we can train the localization model by learning from multiple other available projects.

Implications for researchers. First, JIT defect localization is a real problem which can be the second phase after JIT defect identification. Solely investigating JIT defect identification might be insufficient, since developers cannot identify where is the defect. We believe that JIT defect localization can promote the use of JIT defect identification in practice. Second, software naturalness is effective for JIT defect localization. Currently, we build a simple solution using N-gram. Further advanced methods are expected for improving the performance. Third, both prior clean code and prior buggy code are potentially useful for JIT defect location. Prior clean code can be used to detect the defect which is surprising to clean code, while prior buggy code can be used to detect the defect which is similar to prior buggy code.

6 Threats to Validity

Threats to internal validity relate to potential errors in our implementation. First, one potential threat to validity is the potential errors in our modeling implementation. To mitigate the threat, we use and enhance the source code from a previous study to implement the N-gram modeling on source code [24]. We also double-checked the implementation and fully tested our code, still there could be errors that we did not notice.

Threats to external validity relate to generalizability of our results. Although we have analyzed 177,250 software changes from 14 open-source Java software projects, we cannot claim the generality of our observations to projects written in other programming languages. Instead, the key message of this paper is that there are many Java datasets where our observations are statistically significant. When applying our approach to projects written in other programming languages, some steps (e.g., comments removing and code tokenization) should be carefully adapted. Further investigation of even more projects including projects written in other programming languages is needed to mitigate this threat.

Threats to construct validity relate to the suitability of our evaluation. One potential threat is that we use MRR, MAP and Top-k accuracy as the performance measures, and use Wilcoxon signed-rank test to investigate whether the improvement of our proposed model over baselines is significant. MRR, MAP and Top-k accuracy have been widely used in past software engineering studies [34]–[37], [52]. The Wilcoxon sign-rank test has also been used in many mining software repository studies. Thus, we believe we have little threats to the validity on evaluation measures. Additionally, the evaluation for JIT defect localization is conducted on identified-buggy or all-buggy changes. The clean changes that are wrongly identified as buggy changes (i.e., false positives in JIT defect identification) are ignored. Because in these changes, there is no buggy lines and we cannot calculate the MRR, MAP and Top-k accuracy. This issue may represent a construct validity of our study.

7 Related Work

We divide our related work into three parts: defect localization, software naturalness and Just-in-time (JIT) defect identification.

7.1 Defect Localization

Due to the importance of defect localization, researchers continue to propose various techniques that can be divided into two main categories, i.e., information retrieval (IR) based techniques and spectrum-based techniques.

Information retrieval based method. Lukins et al. [2] proposed an LDA based technique for automated bug localization. They build the localization model by performing

an LDA analysis on source code document collection (e.g., comments and identifiers) and bug reports (e.g., bug title and description). The results indicate that this LDA-based technique outperforms an LSI-based technique by Poshyvanyk et al [53]. Rao and Kak [54] conducted an empirical study by comparing five generic text models, i.e., the Unigram Model (UM), the Vector Space Model (VSM), the Latent Semantic Analysis Model (LSA), the Latent Dirichlet Allocation Model (LDA), and the Cluster Based Document Model (CBDM). They found that simple text models such as UM and VSM are more effective at correctly locating the relevant buggy files as compared to more sophisticated models such as LDA. Saha et al. [34] proposed a new localization technique by combining the bug reports and the structure of code files. Wang and Lo [55] proposed an integrated technique for bug localization by combining version history, similar reports, and structure together.

Spectrum-based based techniques. Jones and Harrold [7] proposed the Tarantula technique that uses a suspiciousness score to locate buggy elements by using the pass/fail information of test cases, the entities that were executed by the test case and the source code for the program under test. Abreu et al. [8] used another similar coefficient formula called Ochiai from the biology domain. They achieved a better performance than Tarantula. Xie et al. [56] conducted a theoretical investigation on the effectiveness of the risk evaluation formulas for spectrum-based localization method and found some formulas outperform others among the 30 studied formulas. Based on this theoretical framework, Xie et al. [57] subsequently analyzed the effectiveness of genetic programming based risk evaluation formulas proposed by Yoo et al. [58] for defect localization.

Our work is inspired by the above-mentioned studies but differs in the usage timing. Both IR-based and Spectrum-based techniques perform the localization based on defect symptoms. IR based methods rely on analyzing the textual description in bug reports, while spectrum-based techniques rely on analyzing program spectrum from actual use of the software system. These techniques are employed long after a defect is discovered. Our work aims to perform defect localization at code check in time–serving as an early quality control step which can complement the current localization techniques.

7.2 Software Naturalness

Software naturalness was originally proposed by Hindle et al [18]. It refers to the intuition that programming languages are highly repetitive. Such repetitiveness can be captured by statistical language models which are originally from the natural language processing (NLP) field. Based on this observation, researchers have leveraged software naturalness for many software engineering tasks. For example, Hindle et al. [18] developed a code completion tool for Java by using software naturalness. Raychev et al. [59] used the APIs to build the language model for code completion. Tu et al. [23] proposed the "cache" mode to improve the performance of code completion by using the locality of software. Allamanis et al. [60] proposed a framework that learns the style of a codebase, and suggests revisions to improve stylistic consistency using software naturalness. Hellendoorn and Devanbu [24] proposed an enhanced language modeling toolkit for source code modeling, they found that carefully adapting N-gram models for source code can outperform deep-learning models.

The most similar papers to ours are the ones that use software naturalness to detect bugs or syntax errors. There are three most similar papers:

Ray et al. [19] proposed a line-level defect prediction approach at the release level using software naturalness. They found that buggy code lines are more "unnatural" than clean code lines, and this observation can be used for enhancing the effectiveness of static bug finding tools (e.g., FindBugs and PMD). Campbell et al. [61] proposed a Java syntax-error locator using an N-gram model. Their approach is trained on the prior versions of a project. Their results show that they can effectively enhance a compiler's ability to locate syntax errors. Based on their observation, Santos et al. [62] proposed the detection and correction of syntax errors using software naturalness. Their approach is trained on clean source code and is also evaluated on many specific revision pairs. Their results show that their approach can locate and suggest corrections for syntax errors.

Our work is inspired by the three aforementioned studies and we do not aim to propose a more advanced approach to outperform these methods. Instead, we aim to adopt their idea for a new framework, i.e., JIT defect identification and localization. Like our work, these previous studies can identify buggy or syntax-error lines using software naturalness. However, we each tackle a different problem. The main differences are as follows.

Ray et al. [19]'s work aims to complement the static defect identification problem for a release, while our work aims to identify the defects in software changes just-in-time. Campbell et al. [61] and Santos et al. [62] aim to detect and correct syntax errors. Our work differs from them in two aspects. First, we aim to identify defects but they are focusing on syntax errors. Second, they do not focus on software changes. Although Santos et al. [62] collected many specific revision pairs from IDE events data (the revision directly prior to a failed compilation due to a syntax error, and the revision immediately following the successful compilation) for evaluating the detection and correction of syntax errors, they only used the revision pairs as the ground truth and they did not conduct the JIT analysis by each change. In summary, different from the above-mentioned studies, we aim to address the JIT defect localization problem on changed content for software changes which is a follow up for any JIT defect identification approach. The timing at which each work will be used is different. Our work aims to identify and locate defects just-in-time when a change is submitted.

7.3 JIT Defect Identification

Researchers has proposed various methods for JIT defect identification. Mockus and Weiss [63] proposed a method for assessing the risk that changes introduce defect in a telecommunication system. Kim et al. [64] proposed an approach to predict the buggy entities and files from cached history at the moment a fault is fixed. Furthermore, Kim

et al. [65] proposed a model for classifying a change as clean or buggy by using various change features. Kamei et al. [9] performed a large-scale empirical study of JIT defect identification using effort-aware evaluation. Shihab et al. [11] conducted an industrial study to better understand risky changes. Yang et al. [13], [14] proposed to use more advanced modeling techniques for JIT defect identification, such as ensemble learning and deep learning. Additionally, many studies investigated the comparison between supervised and unsupervised modeling methods for JIT defect identification [12], [15], [16].

Recently, Nayrolles and Hamou-Lhadj [66] proposed a two-phase approach (called CLEVER) for intercepting risky commits using code clone detection. The first phase is to assess the likelihood that an incoming commit is risky or not. The second phase is to use clone detection to suggest fixes when clone code is detected from the risky commit in the first phase. As a result, 66.7% of the suggested fixes were accepted by developers at industry. Comparing their work to this paper, the similarity is that both of us conducted a two-phase analysis to support how to fix the commits. The difference is the aim. They aim to suggest the fixes from prior similar fixes at the moment clone code is detected. We aim to locate the most suspicious lines in any buggy change to save the code inspection effort.

Pascarella et al. [67] proposed a fine-grained JIT defect identification approach which can identify defective files within changes. They actually conducted a file-level defect prediction that focus on the changed files in commits. Different from our work, they did not conduct the integrated analysis on JIT defect identification and localization. Additionally, they focus on file-level, while we focus on line-level.

Despite the success achieved by the above-mentioned techniques, there is no attempt for telling developers where is the most suspicious defective lines after JIT defect identification. Thus, our work aims to be served as the next step of JIT defect identification. In other words, our JIT defect localization phase can be employed when we identify a change as buggy.

8 Conclusion and Future work

In this paper, we propose a two-phase framework, i.e., JIT defect identification and localization. For JIT defect identification phase, we implement a logistic regression based approach using 14 change-level features. For JIT defect localization phase, we use N-gram modeling technique to build a source code language model training on historical clean source code. When given a new change, JIT defect identification phase will identify it as buggy or clean first. If a new change is identified as buggy, JIT defect localization phase will compute an entropy value for each line introduced by the change. Then, we can sort all the introduced lines according to the entropy value. The lines sorted at the top are more likely to be the defect location. To evaluate the effectiveness of our framework, we conduct an empirical study on 14 open source projects with totally 177,250 changes.

In summary, the empirical study results show that: (1) Our framework achieves a reasonable and better performance than the baselines (i.e., PMD and RG) in terms of MRR, MAP and Top-k accuracy. (2) Considering the buggy changes that are correctly identified by our first phase (i.e., identified-buggy changes), our approach can successfully locate the first buggy line in near 3 lines on average (i.e, a MRR of 0.396). And our approach can successfully locate at least one buggy line in top-1 line among 26.5% and in top-5 lines among 54.4%. (3) A buggy model can also be possible for JIT defect localization. However, a clean model is a better choice on average.

Our paper is an important step in studying the effectiveness of JIT defect localization. We envision many future efforts to extend our work for JIT defect localization as well as to improve JIT defect identification (and possibly even crafting approaches that create a feedback loop between these two approaches). For example, further research can investigate whether or not the entropy of the changed lines could be used to enhance (or even replace) the state-of-the-art JIT defect identification approach. Whether or not the JIT defect identification and localization are effective on pull requests needs further investigation, since it is hard to obtain the ground truth whether commits in unaccepted pull requests are buggy or clean using the SZZ algorithm that we leverage in this study. With respect to the language modeling step, further research can investigate whether or not a cache model (i.e., considering the locality of changed content), or a nested model (i.e., combining CP, WP and Cache models) can improve the effectiveness of JIT defect localization. With respect to the granularity, further research can adapt our approach at a wider granularity for the JIT defect localization (e.g., a block-level defect localization by considering a larger context of each change). Finally, additional in depth case studies on JIT defect localization in practice and the perception of developers should be conducted. This can help the community obtain a better understanding of the problem and collect useful information to improve this task further.

Acknowledgment. This research was partially supported by the Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme (DE200100021) and the China Postdoctoral Science Foundation (2019T120517). Additionally, we are very grateful to Hellendoorn and Devanbu [24] and Rosen et al. [10] for sharing their scripts.

References

[1] T. V.-D. Hoang, R. J. Oentaryo, T.-D. B. Le, and D. Lo, "Network-clustered multi-modal bug localization," *IEEE Transactions on Software Engineering*, 2018.

[2] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.

[3] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 14–24.

[4] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in *Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on*. IEEE, 2017, pp. 218–229.

[5] K. C. Youm, J. Ahn, and E. Lee, "Improved bug localization based on code change histories and bug reports," *Information and Software Technology*, vol. 82, pp. 177–192, 2017.

[6] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: statistical model-based bug localization," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 286–295.

[7] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 2005, pp. 273–282.

[8] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[9] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013.

[10] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 966–969.

[11] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 62.

[12] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 157–168.

[13] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*. IEEE, 2015, pp. 17–26.

[14] X. Yang, D. Lo, X. Xia, and J. Sun, "Tlel: A two-layer ensemble learning approach for just-in-time defect prediction," *Information and Software Technology*, vol. 87, pp. 206–220, 2017.

[15] Q. Huang, X. Xia, and D. Lo, "Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction," in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 159–170.

[16] W. Fu and T. Menzies, "Revisiting unsupervised learning for defect prediction," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 72–83.

[17] 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 *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–10.

[18] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 837–847.

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

[20] Q. Huang, X. Xia, and D. Lo, "Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction," *Empirical Software Engineering*, p. In press, 2018.

[21] E. C. Neto, D. A. da Costa, and U. Kulesza, "The impact of refactoring changes on the szz algorithm: An empirical study," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 380–390.

[22] Y. Fan, X. Xia, D. A. da Costa, D. Lo, A. E. Hassan, and S. Li, "The impact of changes mislabeled by szz on just-in-time defect prediction," *IEEE Transactions on Software Engineering*, 2019.

[23] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 269–280.

[24] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 763–773.

[25] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *ACM sigsoft software engineering notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.

[26] S. Kim, T. Zimmermann, K. Pan, E. James Jr *et al.*, "Automatic identification of bug-introducing changes," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE, 2006, pp. 81–90.

[27] 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," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2017.

[28] Y. S. Nugroho, H. Hata, and K. Matsumoto, "How different are different diff algorithms in git?" *Empirical Software Engineering*, pp. 1–34, 2019.

[29] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead Jr, "Mining version archives for co-changed lines," in *MSR*, vol. 6. Citeseer, 2006, pp. 72–75.

[30] Y. Fan, X. Xia, D. Lo, and S. Li, "Early prediction of merged code changes to prioritize reviewing tasks," *Empirical Software Engineering*, pp. 1–48, 2018.

[31] S. F. Chen and J. Goodman, "An empirical study of smoothing techniques for language modeling," *Computer Speech & Language*, vol. 13, no. 4, pp. 359–394, 1999.

[32] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.

[33] D. M. Christopher, R. Prabhakar, and S. Hinrich, "Introduction to information retrieval," *Cambridge University Press*, vol. 151, no. 177, p. 5, 2008.

[34] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 345–355.

[35] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (n)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 476–481.

[36] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Automatic api recommendation using crowdsourced knowledge," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, pp. 349–359.

[37] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 2016, pp. 262–273.

[38] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the imprecision of cross-project defect prediction," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 61.

[39] T. Copeland, "Pmd applied," 2005.

[40] X. Xia, E. Shihab, Y. Kamei, D. Lo, and X. Wang, "Predicting crashing releases of mobile applications," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2016, p. 29.

[41] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu, "Comparing static bug finders and statistical prediction," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 424–434.

[42] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *Acm sigplan notices*, vol. 39, no. 12, pp. 92–106, 2004.

[43] A. E. Hassan and R. C. Holt, "Studying the evolution of software systems using evolutionary code extractors," in *Software Evolution, 2004. Proceedings. 7th International Workshop on Principles of*. IEEE, 2004, pp. 76–81.

[44] F. Hassan, S. Mostafa, E. S. Lam, and X. Wang, "Automatic building of java projects in software repositories: A study on feasibility and challenges," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2017, pp. 38–47.

[45] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "There and back again: Can you compile that snapshot?" *Journal of Software: Evolution and Process*, vol. 29, no. 4, p. e1838, 2017.

[46] F. Wilcoxon, "Individual comparisons by ranking methods," *Breakthroughs in Statistics*, pp. 196–202, 1992.

[47] H. Abdi, "Bonferroni and šidák corrections for multiple comparisons," *Encyclopedia of measurement and statistics*, vol. 3, pp. 103–107, 2007.

[48] N. Cliff, *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[49] J. L. Hintze and R. D. Nelson, "Violin plots: a box plot-density trace synergism," *The American Statistician*, vol. 52, no. 2, pp. 181–184, 1998.

[50] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.
[51] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: bug detection with n-gram language models," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 708–719.
[52] B. Xu, Z. Xing, X. Xia, D. Lo, Q. Wang, and S. Li, "Domain-specific cross-language relevant question retrieval," in *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 413–424.
[53] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, 2007.
[54] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 2011, pp. 43–52.
[55] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 53–63.
[56] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 4, p. 31, 2013.
[57] X. Xie, F.-C. Kuo, T. Y. Chen, S. Yoo, and M. Harman, "Provably optimal and human-competitive results in sbse for spectrum based fault localisation," in *International Symposium on Search Based Software Engineering*. Springer, 2013, pp. 224–238.
[58] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *International Symposium on Search Based Software Engineering*. Springer, 2012, pp. 244–258.
[59] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Acm Sigplan Notices*, vol. 49, no. 6. ACM, 2014, pp. 419–428.
[60] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 281–293.
[61] J. C. Campbell, A. Hindle, and J. N. Amaral, "Syntax errors just aren't natural: improving error reporting with language models," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 252–261.
[62] E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, and J. N. Amaral, "Syntax and sensibility: Using language models to detect and correct syntax errors," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 311–322.
[63] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.
[64] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 489–498.
[65] S. Kim, E. J. Whitehead Jr, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.
[66] M. Nayrolles and A. Hamou-Lhadj, "Clever: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects," in *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 2018, pp. 153–164.
[67] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *Journal of Systems and Software*, 2018.