Leveraging Fault Localisation to Enhance Defect Prediction

Jeongju Sohn
KAIST
Daejeon, Korea
kasio555@kaist.ac.kr

Yasutaka Kamei
Kyushu University
Kyushu, Japan
kamei@ait.kyushu-u.ac.jp

Shane McIntosh
University of Waterloo
Waterloo, Canada
shane.mcintosh@uwaterloo.ca

Shin Yoo
KAIST
Daejeon, Korea
shin.yoo@kaist.ac.kr

***Abstract*—Software Quality Assurance (SQA) is a resource constrained activity. Research has explored various means of supporting that activity. For example, to aid in resource investment decisions, defect prediction identifies modules or changes that are likely to be defective in the future. To support repair activities, fault localisation identifies areas of code that are likely to require change to address known defects. Although the identification and localisation of defects are interdependent tasks, the synergy between defect prediction and fault localisation remains largely underexplored.**

We hypothesise that modifying code that was suspicious in the past is riskier than modifying code that was not. To validate our hypothesis, in this paper, we employ fault localisation, which localises the root cause of a program failure. We compute the past suspiciousness score of code changes to each fault, and use those scores to (1) define new features for training defect prediction models; and (2) guide the next actions of developers for a commit labelled as fix-inducing. An empirical study of three open-source projects confirms our hypothesis. The new suspiciousness features improve F1 score and balanced accuracy of Just-In-Time (JIT) defect prediction models by 4.2% to 92.2% and by 1.2% to 3.7%, respectively. When guiding developer actions, past code suspiciousness successfully guides developers to a defective file, inspecting two to nine fewer files on average, compared to the baselines based on previous findings on past faults. These results demonstrate the potential of synergies of fault localisation and defect prediction, and lay the groundwork for explorations of that combined space.

***Index Terms*—defect prediction, fault localisation, search-based software engineering, software quality assurance**

I. Introduction

Since exhaustive testing is impractical (if not impossible), software organizations must take pragmatic and cost-effective approaches to Software Quality Assurance (SQA). To focus SQA detection activities, defect prediction [1], [2] has been proposed to prioritize modules by their likelihood of being defective. Modern so-called "Just-In-Time" (JIT) defect prediction models identify risky code changes, i.e., code changes in a fix-inducing commit, rather than defective modules. Targeting risky changes fits nicely within a change-based development model, raising warnings about changes while the context of the changes is still fresh in the mind of developers [3].

To support SQA repair activities, fault localisation [4] has been proposed to prioritise areas within the codebase that are likely to require modification to address a defect. Fault localisation ranks program elements by their likelihood of being implicated in the fix for a given program failure (suspiciousness). Information Retrieval-based Fault Localisation (IRFL), one of the most actively studied approaches to fault localisation, exploits the lexical similarity between code and bug reports to localise faulty code [5]–[7]. Unlike the most dynamic FL techniques that require test execution, IRFL can be applied as soon as bug reports are available.

Despite being interdependent tasks, the literature on (JIT) defect prediction and fault localisation have largely developed independently of one another. Indeed, both fault localisation and defect prediction strive to support quality assurance activities with different timing—defect prediction and fault localisation occur before and after program failures have been identified, respectively. Exploiting synergies between the two distinct areas may yield more accurate and actionable insights. Early work by Sohn and Yoo [8], [9] shows that features that have typically been used in defect prediction (i.e., code and change features) can improve the performance of fault localisation. In this paper, we set out to explore the inverse. More specifically, we set out to address the following question:

Can insights from fault localisation be leveraged to enhance JIT defect prediction?

To address this central question, we quantify the suspiciousness of past code using IRFL. Computing the past suspiciousness of code using fault localisation is a natural choice because it shares a similar goal with JIT defect prediction, i.e., identifying risky code. We conjecture that code that was suspicious in the past will remain suspicious for some time and warrants additional scrutiny when being modified. We first evaluate whether using past suspiciousness measures of code improve the performance of JIT defect prediction models. Then, we investigate whether these past suspiciousness measures aid in guiding the repair activities of developers for risky commits. To evaluate our approach of using past suspiciousness of code in defect prediction, we perform an empirical study of three open-source projects. In addition to that empirical study, this paper makes the following contributions:

- We define new defect prediction features based on the prior suspiciousness of code according to fault localisation. Using these features, we observe that the F1 score of JIT models improves by 4.2% to 92.2%. We also observe a small increase in Balanced Accuracy of 1.2% to 3.7%.

- We use the prior suspiciousness of code to guide the next action of developers after a commit is predicted to be risky. The experimental results show that by inspecting files modified in a commit in descending order of their suspiciousness, developers can examine the first defective file, on average, two to nine files faster.

The remainder of this paper is organised as follows. Section II introduces our hypothesis on leveraging the past suspiciousness of code changes in a commit and the two approaches that we took to verify this hypothesis. Section III presents three research questions and describes the experimental setup for the empirical study. Section IV discusses the results and Section V presents the threats to validity. Section VI describes the related work, and lastly, Section VII concludes our paper.

II. FL2DP: Fault Localisation to Defect Prediction

Defect Prediction (DP) aims to predict whether a given change is fix-inducing or not, before actually executing any test. Fault localisation, on the other hand, aims to predict the location of a fault whose existence has been already confirmed by a failing test execution. Although both techniques have been widely studied, the interplay between two techniques has not been thoroughly investigated. Existing work has investigated how features frequently used for defect prediction can improve accuracy of fault localisation [8], [9], but the opposite direction of using fault localisation for better defect prediction has not been explored. This section motivates the way we make use of previous faults via fault localisation for better DP.

A. The Impact of Past Faults in Defect Prediction

Kim et al. showed the localities of faults: faults appear together instead of individually [10]. The reasoning is that developers make fix-inducing changes due to the incorrect understanding of the changes, and therefore are likely to introduce more than one fault in the area of misunderstanding, either directly or indirectly (e.g., via the propagation of changes). The authors introduced various types of localities: temporal, spatial, and changed or new-entity. Despite their differences, they all assume that the area that has been related to faults, either was defective or changed along, is riskier. Rahman et al. followed this up by comparing these localities to each other and reported that temporal locality, which presumes that recently fixed code is more defect-prone, performs the best for defect prediction [11].

Another way that previously defective elements become more defect prone in the future is if fixing previous faults actually resulted in new ones. Mockus and Weiss reported that changes that fix defects are riskier than the other types of changes, such as new features [12]. Shihab et al. performed an industrial case study on the risk of software changes [13]: their study showed that the more bug-fixing changes were made on a file, the riskier the file is. Combined, these results suggest that areas of source code that contained previous faults tend to be more defect prone in the future.

We propose fault localisation techniques as a unified framework that can capture various aspects of previous faults studied in the context of defect prediction. Instead of looking at binary property of whether a file has been defective in the past or not, or whether a file has been a target of bug fix or not, we measure how *suspicious* the changed file would be with respect to previous faults, using a specific fault localisation technique. This way, we capture degrees of suspiciousness instead of the binary property of being faulty or not.

In the software life-cycle, there are two processes that aim to locate the code related to program failure: Fault Localisation (FL) and Defect Prediction (DP). Both FL and DP are designed to assist developers with testing and debugging. Their tasks can be defined as follows:

- **Fault Localisation (FL)**: *locate* the source code that caused the observed program failure
- **Defect Prediction (DP)**: *predict* whether the changed source code will result in a program failure

If DP is an *a priori* detection of faults, FL is an *a posteriori* detection of faults, i.e., knowing where the fault is after the fact. In the software development timeline, these two tasks will interleave with each other. Consequently, if there are temporal and spatial locality in the occurrence of faults, we posit that FL results for previous faults can help with DP for the current change. More specifically, we formulate the following hypothesise:

> **Hypothesis:** If a commit modifies code that was suspicious in the past (FL), the commit is more likely to introduce faults than those that were not (DP).

To validate this hypothesis, we define new features based on the results of fault localisation on previous faults and investigate whether these features can improve the performance of defect prediction. We also study how the suspiciousness of code computed from past fault localisation can make the DP results more actionable for developers. The remainder of this section introduces the new features and describes how we use them to enhance the actionability of DP results.

B. IR based Fault Localisation as Defect Prediction Feature

Table I shows the input features used for our defect prediction model. The first 13 features are features that have been widely studied for defect prediction and are taken from existing work [3]. We have excluded three review features studied by McIntosh and Kamei, as our choice of benchmark (see Section III-B) did not adopt an explicit code review process. The remaining three features are based on IRFL.

While a diverse range of fault localisation techniques have been studied [5], [7]–[9], [14]–[17], we focus on IRFL because of two reasons: its non-dynamic nature, and its flexibility. The intuition behind IRFL is that a bug report, and the part of the source code that is the root cause of the reported bug, are likely to share a strong lexical similarity with each other. Based on this intuition, IRFL adopts various textual similarity metrics developed in the field of Information Retrieval (IR) to rank

program elements according to their similarity to the given bug report: the given bug report becomes the *query*, which is used to search the *documents* that are program elements. Consequently, it does not require any dynamic analysis, apart from the occurrence of the bug that resulted in the original bug report. This makes IRFL an ideal candidate for a DP feature, since DP is typically performed *before* the test execution.

By flexibility of IRFL, we refer to the fact that textual similarity can be computed between any bug report and any program element, regardless of the timeline. Our hypothesis is that modifying code that was suspicious for historical faults poses higher risk. However, if we were to compute the exact suspiciousness of each program element at the time of each historical fault, it would result in non-trivial analysis cost even if the FL technique of choice is a non-dynamic one such as IRFL. To avoid this, we approximate the past suspiciousness by taking the textual similarity between *historical* bug reports and the *current* program elements that have been modified and submitted for defect prediction. We posit that there will be sufficient continuity in the set of lexical tokens that appear in the same program element such as a file or a specific function. This lexical continuity, coupled with the temporal and spatial locality of fault [10], allows us to avoid the inherently noisy and costly process of tracing change history from current changes to the locations of previous faults.

TABLE I
16 FEATURES USED TO TRAIN DEFECT PREDICTION MODELS

	Property	Description
Size	Lines Added	The number of added lines in a commit
	Lines Deleted	The number of deleted lines in a commit
Diffusion	Subsystem	The number of modified subsystems in a commit
	Directory	The number of modified directory in a commit
	File	The number of modified files in a commit
	Entropy	The spread of modified lines across files in a commit
History	Changes	The number of changes made to the modified files in past
	Developers	The number of developers who have changed the modified files in a commit in the past
	Age	The time interval to the last changes on the modified files
Experience	Prior changes	The number of prior changes to the modified files the authors participated
	Recent changes	The number of prior changes to the modified files the authors participated weighted by the time interval between changes
	Subsystem changes	The number of prior changes to the modified directories the authors participated
	Awareness	The fraction of prior changes to the modified directories that the authors participated
Bug report	$sim2r_{sum}$	The sum of similarities between code changes in a commit and recent bug reports
	$sim2r_{max}$	The maximum of similarities between code changes in a commit and recent bug reports
	$sim2r_{mean}$	The arithmetic mean of similarities between code changes in a commit and recent bug reports

1) Vector Space Model for IRFL: We use IRFL technique based on Vector Space Model (VSM) [18] to compute the similarity between code changes made in a commit and previous bug reports [4]. After the standard text preprocessing that includes text normalisation[1], stopword removal, and word stemming, both the query (i.e., the bug report) and the documents (i.e., program elements, such as source files) are represented as fixed-length weight vectors, where each weight corresponds to a unique word (term). The typical means of obtaining weights for each term is to compute term frequency-inverse document frequency (*tf-idf*). Given a term t, a document d, and the set of all documents, D, let $f_{t,d}$ be the number of times t appears in d. *tf-idf* is defined as follows:

$$tf(t,d) = \log(1 + f_{t,d}) \tag{1}$$

$$idf(t,D) = \log(\frac{|D|}{|\{d \in D | t \in d\}|}) \tag{2}$$

$$tf-idf(t,d,D) = tf(t,d) \cdot idf(t,D) \tag{3}$$

Intuitively, $tf(t,d)$ captures the degree to which a term t is coupled with a document d (i.e., the frequency of appearance of t in d), whereas $idf(t,D)$ captures the degree of how narrow the coupling between t and documents is (i.e., higher $idf(t,D)$ means most of documents in D contain t, whereas lower $idf(t,D)$ means only a few documents in D contain t). A *tf-idf* weight is simply the product of TF and IDF of a given term; a document d, or a query q, is represented as a fixed-length vector of *tf-idf* weights, each of which is the weight for a term in the corpus D. Finally, the similarity between a document and a query can be computed as the cosine similarity between two *tf-idf* vectors.

2) Feature Engineering for IRFL Scores: For IRFL, we have multiple documents (i.e., source code files) to match against a single query (i.e., a bug report). However, for its use in the context of Just-In-Time DP, we would like to measure how similar the single commit under consideration is to a range of potentially related previous faults. This results in a single document (i.e., the commit) and multiple queries (i.e., the previous faults that are potentially relevant). Both factors affect how we convert an IRFL score (i.e., similarity) to a defect prediction feature.

The single document situation means that the idf's only information is whether a term is in documents or not, which can be covered by tf. As a result, we fix the idf to 1, using only the tf for actual computation of the similarity. Given that each commit should ideally contain only a single change, we believe that using term frequency to represent its topic is an acceptable choice. Even if the commit under consideration addresses multiple issues, we expect the range of topics to be much narrower and more focused compared to all possible semantic topics in the entire system. Note that the single document issue only applies when we try to convert the IRFL score into a feature for defect prediction of a single commit under consideration. If the commit is predicted to be defective, we would like to *localise* the cause of concern for actionability, for which we consider different files in the single commit as separate documents. See Section II-C for more details.

The issue with multiple queries is that we cannot ascertain which previous fault is the most relevant to the commit under

[1]We normalise camelCase identifiers into its constituent subtokens.

consideration. To resolve this, we focus our analysis on the bug reports that have been created within the predefined time window (based on the temporal locality assumption). Since there are often multiple bug reports within the time window, multiple similarity values are calculated for the given commit. We aggregate these similarity values using three different methods and define one feature for each: arithmetic mean ($sim2r_{mean}$), sum ($sim2r_{sum}$), and maximum ($sim2r_{max}$).

Here, c denotes the commit under consideration, and r a bug report submitted to the same project. By defining a time window, we first identify the set of bug reports ($R(c, w)$) that have been submitted during the time window as follows:

$$R(c, w) = \{r | timestamp(c) - timestamp(r) \leq w\} \quad (4)$$

Now, let $cl(c)$ be the set of lines modified by c, and M be one of the aggregation methods, $\{\text{arithmetic mean}, \text{sum}, \text{max}\}$. The similarity based DP feature for the given commit c under M is defined as follows:

$$sim2r_M(cl(c), w, R) = M(\{sim(cl(c), r) | r \in R(c, w)\}) \quad (5)$$

That is, $sim2r_M(cl(c), w, R)$ is either arithmetic mean, sum, or maximum of similarities between the changed lines in the given commit and bug reports submitted within the given time window from c. We use cosine similarity for sim.

C. Actionability

A defect prediction result is *actionable* if it can guide the developer on how to handle the potentially fix-inducing commit. In an ideal case, all changes in a commit will be tightly coupled to each other, resulting in all files modified by a fix-inducing commit being responsible for the induced defect. In practice, however, this is not always the case: a fix-inducing commit often contains changes unrelated to the induced fault. As a result, without knowing which file is defective, developers may waste their effort inspecting non-defective files. Given this consideration, we propose to use the similarity to previous bug reports to guide the developer.

Our approach presumes the temporal locality of faults: similar faults may appear again within a short time frame. We exploit this for actionability by aiming to *localise* the fault among the files modified by the given commit. Since there is no information about what exactly the induced fault will be at the time of defect prediction, we use the previous bug reports as an approximation, using the temporal locality assumption: we run IRFL on the files changed by the given commit against previous bug reports, i.e., $R(c, w)$. The suspiciousness of a file f modified by a fix-inducing commit c is computed as follows:

$$susp(f, c, w, R) = \sum_{r \in R(c,w)} sim(cl(c, f), r) \quad (6)$$

Here, $cl(c, f)$ denotes the set of lines in file f that have been changed by the commit c. The final suspiciousness score ($susp$) of a file f is the sum of the similarities between the changes of the file and previous bug reports within the time window w. Note that $sim2r_M$ is used to denote the defect prediction feature, which represents the commit under consideration, whereas $susp(f, c, w, R)$ denotes the individual IRFL suspiciousness of each file modified by the commit c, in the context of actionability. We hereafter refer to $susp(f, c, w, R)$ as $susp_f$ for the sake of brevity.

III. Experimental Setup

In this section, we present our research questions and the corresponding experimental setup for our empirical study.

A. Research Questions

We formulate three research questions to evaluate the utility of the past suspiciousness of code changes in a commit when predicting its defect-proneness.

RQ1. Feature Analysis: *Do $sim2r_M$ features capture distinct and useful features of fix-inducing commits?* Before using $sim2r_M$ features introduced in Section II-B2, we investigate whether these features can capture distinct and useful properties of fix-inducing commits.

Firstly, to estimate the degree to which our features are distinct, we perform Spearman correlation analysis on the 16 features in Table I. If $sim2r_M$ features capture new dimensions that have not yet been explored, they will have a weak correlation to existing JIT DP features. Secondly, to show that our features can be useful in defect prediction, we perform Mann-Whitney U-tests on the $sim2r_M$ features of fix-inducing commits and non-inducing commits ($\alpha = 0.05$). Since $sim2r_M$ features assume that defective changes are more similar to past faults than non-defective changes are, we expect fix-inducing commits to have larger values of $sim2r_M$ features. We also compare the feature importance values in the learnt defect prediction models between the $sim2r_M$ features and the other 13 features to assess how useful the $sim2r_M$ features are compared to the other JIT-DP features.

RQ2. Effectiveness: *Can the use of $sim2r_M$ features improve the performance of defect prediction models?* To answer RQ2, we compare the performance of defect prediction models trained with and without $sim2r_M$ features. Before comparing them, however, we first investigate the relationship between the effectiveness of $sim2r_M$ features and the duration of the time window. $sim2r_M$ features aggregate the similarities between code changes and bug reports that are created within a fixed time window. Thus, the effectiveness of $sim2r_M$ features can vary depending on the choice of the time window. We use four different values for the time window, 30, 60, 90, and 120 days, and inspect how the value of the time window affects the benefit of using $sim2r_M$ features. We then go back to the initial question and compare the performance of models with and without $sim2r_M$ features, using the best time window among the four. We run Mann-Whitney U-test on the pairs of the DP models with and without $sim2r_M$ features and measure Cliff's delta effect size for these pairs. We deem the improvement from using $sim2r_M$ features is statistically significant if the p-value of the U-test is ≤ 0.05, and the effect size is at least medium ($\delta > 0.33$) [19].

RQ3. Actionability: *Can $susp_f$ guide the action of developers for fix-inducing commits?* To validate the actionability of past suspiciousness of code ($susp_f$), we compare the effort spent before inspecting the first defective file when examining files modified in a commit in descending order of their $susp_f$ scores with the effort when examining the files randomly. If $susp_f$ scores computed per file are truly actionable, they will allow developers to discover a defective file with less effort.

In addition to the random inspection, we use closed bug count (cnt_{CB}) as another baseline of $susp_f$. Closed bug count of a file is the number of closed bugs that modified the file. Here, *closed* bug denotes the bug that has been closed before the commit. Previous studies used the closed bug count to sort files in order of their defect-proneness [11], [20]. Results of these studies show that simply ordering files in descending order of their closed bug counts could be as effective as more sophisticated defect prediction techniques.

Compared to previous work, which utilises only the ground-truth of defective files, we use past suspiciousness of files that were non-defective in the past along with past suspiciousness of files that were defective. To verify that the improvement gained from using $susp_f$ does not solely come from computing the suspiciousness scores of defective files, but rather from taking both defective and non-defective files into consideration, we select a variant of cnt_{CB} called sim_{CB} as the last baseline. sim_{CB} stands for the similarity to past closed bugs; it adds up the suspiciousness scores of a file to each past fault for which it was responsible instead of counting them. Consequently, if the use of $susp_f$ also outperforms the use of sim_{CB}, we can conclude that including files that were non-defective in the past, in addition to past defective files does contribute positively.

Unlike cnt_{CB} and sim_{CB}, $susp_f$ does not require bug reports to be closed for computation, as it uses suspiciousness scores of code. Nevertheless, we take the closed bugs as inputs for $susp_f$ by default. To fully exploit $susp_f$, we investigate how the effectiveness of $susp_f$ changes as more reports become available for inspection. For this, we conduct another experiment that compares the effectiveness of $susp_f$ with different sets of reports: a set of both open and closed bug reports, and a set of reports of all types.

TABLE II
THREE OPEN-SOURCE BENCHMARKS: THE TABLE ON THE LEFT DESCRIBES THE COLLECTED COMMITS, AND THE TABLE ON THE RIGHT SHOWS THE AVERAGE NUMBER OF BUG REPORTS COLLECTED USING FOUR DIFFERENT TIME WINDOWS.

Project	# of commits	# of non-inducing commits	# of fix-inducing commits	Collection Period	Avg. # of bug reports in diffrent time window (days) 30	60	90	120
Lang	3,929	3,697	232	2002/07 – 2019/10	3.15	5.50	7.59	9.74
Math	4,810	4,325	485	2003/05 – 2019/10	4.44	8.12	11.78	15.43
Closure	1,948	1,848	100	2009/11 – 2013/12	7.46	14.60	21.64	28.51

B. Subjects

We evaluate our approach of leveraging the past suspiciousness of code using Defects4J [21], a real-world fault dataset frequently studied in fault localisation. We use Defects4J v.1.5.0, which contains six open-source projects; these projects manage their issue reports using different issue trackers. Among the six projects, we select three projects, Commons-Lang, Commons-Math, and Closure Compiler, for the evaluation. The main criterion of this selection is how structured an issue report is. If issue reports are written freely, there will be many variables to consider when parsing them, e.g., whether to include comments. Thus, we prefer projects that use an issue tracker with a strict format. Lang and Math adopt Jira [22], an issue tracker that stores issues in a highly structured format and is designed for use by agile development teams. Issues in Closure that are covered by Defects4J are stored in Google Code Archive. As issue reports managed by Git issue tracker are less well structured, we exclude the projects that use the Git issue tracker, leaving only Lang, Math, and Closure.

We use the SZZ algorithm to identify fix-inducing commits [23]. Instead of implementing our own, we use the open-source implementation of SZZ algorithm [24]. To filter out false-positives, we adopt six of the eight filters applied by McIntoch and Kamei [3]. We omit the two filters that exclude the commits that either fix or induce too many faults because they were too aggressive, leading to several false-negatives, when they were applied to our dataset. Table II shows the final number of commits for each category, either fix or non-inducing, and the associated commit collection period.

C. Training Defect Prediction Models

For training defect prediction models, we use Random Forest (RF), which showed good-enough performance in previous studies of defect prediction [25]–[30]. We employ `scikit-learn` [31] for Random Forest and set the number of trees as 200. For other parameters of RF, we use the default values provided in *scikit-learn*. We have decided to use the default parameter setting, as our initial grid-search based hyperparameter tuning has failed to improve performance consistently and significantly across performance metrics and studied subjects compared to the default settings. We would recommend careful parameter tuning in case the proposed technique is applied to a real project with stable history.

For training and test data sets, we sort the collected commits in their chronological order, i.e., commit time, and sequentially divide them into five folds. We train defect prediction models for each fold using all commits prior to the target fold. Since the first fold, i.e., fold zero, does not have any preceding fold to train defect prediction models, we exclude fold zero from the evaluation. Figure 1 describes how the fix-inducing commit rate changes over time. As software becomes more mature, the number of fix-inducing commits decreases dramatically. For each fold, we repeat the training 30 times due to the randomness of Random Forest.

D. Evaluation Metrics

To answer RQ2, we compare the performance of defect prediction models trained with and without $sim2r_M$ features and examine whether using $sim2r_M$ features improves the performance of defect prediction models. We use five metrics

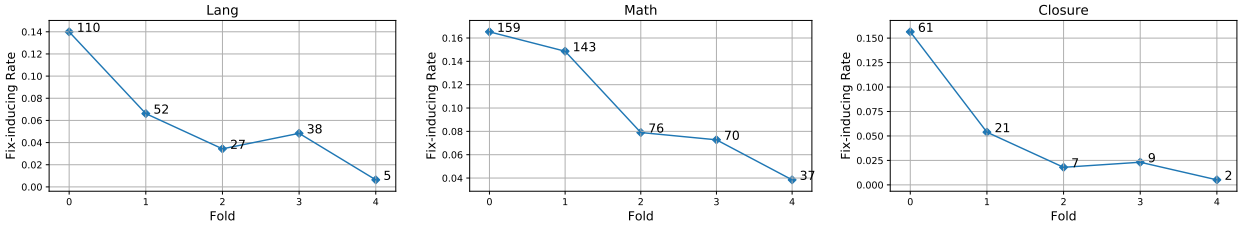


Fig. 1. Fix-inducing rate per fold: the actual number of fix-inducing commits is written next to each point. For all three project, the fix-inducing rate decreases gradually as the projects become more mature.

that are frequently adopted to evaluate the performance of defect prediction models: ROC-AUC, Accuracy, Balanced Accuracy, F1 score, and PR-AUC.

ROC-AUC measures the capability of a model distinguishing two classes, in our case, fix-inducing and non-inducing commits. Accuracy is the overall rate of correctly classified instances, while Balanced Accuracy is the average rate of correctly classified instances of each class. F1 score is the harmonic mean of Precision and Recall, and PR-AUC is the area under the curve of Precision and Recall; it measures the trade-off between these two as a threshold changes.

RQ3 concerns whether $susp_f$ can direct developers to defective files. To answer this, we compare the effort of developers spent before meeting the first defective file. We assume that the effort is proportional to the number of non-defective files examined before the first defective one and use the number of those non-defective files as the effort. As mentioned in Section III-A, we compare $susp_f$ with three baselines: the random, the closed bug count (cnt_{CB}), and the closed bug similarity (sim_{CB}). Among these baselines, the random approach is inherently stochastic and requires to be repeated to generalise its outcome. Instead of repeating the random approach, we compute the expected rank of the first defective file and use it as an alternative. Assuming that the likelihood of each file to be inspected is uniform, we define the expected rank of the first defective file as below:

$$exp(f,c) = \frac{\sum_{rank=1}^{M-N+1} rank}{M},\ M = |F_c|, N = |GT_c| \qquad (7)$$

As Equation (7) shows, the expected rank of the first defective file is the arithmetic mean of all possible ranks of defective files. Here, M is the total number of files modified in a commit c, and N is the number of those files whose changes induce the fault. We also use this expected rank as an alternative for the cases where all the modified files have the same ranking by the other approaches.

In addition to the number of inspected files, we use $acc@n$ metric, which is often adopted for the evaluation of fault localisation, for the effort evaluation. $acc@n$ counts the number of fix-inducing commits whose ranking contains at least one defective file within the top n. For example, if $n = 5$ and $acc@n = 10$, it means that at least one defective file has been successfully located within the top five for ten fix-inducing commits. For the random approach, we compute $acc@n$ by counting the number of fix-inducing commits in which the expected rank is smaller or equal to n.

TABLE III
MANN-WHITNEY U-TEST ON $sim2r_M$ FEATURES BETWEEN FIX-INDUCING AND NON-INDUCING COMMITS WITH A STATISTICAL SIGNIFICANCE OF 0.05. P-VALUES SMALLER OR EQUAL TO 0.05 ARE IN BOLD TEXT. F STANDS FOR FOLD.

		$sim2r_{sum}$				$sim2r_{mean}$				$sim2r_{max}$			
Proj	F	30	60	90	120	30	60	90	120	30	60	90	120
Lang	0	0.12	**0.00**	**0.01**	**0.02**	**0.00**	**0.00**	**0.00**	**0.00**	0.15	0.24	0.46	0.56
	1	0.55	0.22	0.16	0.29	0.41	0.14	0.08	0.10	0.75	0.75	0.63	0.74
	2	0.49	0.21	0.09	0.14	**0.01**	**0.00**	**0.00**	**0.00**	0.23	0.13	0.13	0.20
	3	0.18	0.07	**0.04**	**0.01**	**0.01**	**0.00**	**0.00**	**0.00**	**0.05**	**0.01**	**0.01**	**0.01**
	4	0.39	0.13	0.06	0.06	0.73	0.54	0.50	0.50	0.79	0.61	0.63	0.74
	all	0.07	**0.00**	**0.00**	**0.00**	**0.03**	**0.00**	**0.00**	**0.00**	0.28	0.25	0.29	0.45
Math	0	1.00	1.00	0.97	0.95	0.86	**0.00**	**0.00**	**0.00**	1.00	0.71	0.43	0.29
	1	0.15	**0.00**	**0.01**	**0.01**	**0.00**	**0.00**	**0.00**	**0.00**	0.53	0.41	0.67	0.70
	2	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.21	0.53	0.92	0.74
	3	0.47	**0.01**	**0.00**	**0.01**	0.24	**0.02**	**0.00**	**0.00**	0.95	0.97	1.00	1.00
	4	**0.02**	0.09	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.07	**0.02**	**0.03**
	all	0.80	0.14	0.11	0.10	**0.00**	**0.00**	**0.00**	**0.00**	0.96	0.84	0.94	0.90
Closure	0	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.11	0.74	0.93	0.98
	1	0.48	0.52	0.42	0.46	0.36	0.42	0.32	0.38	0.93	0.97	0.97	0.95
	2	0.22	0.23	0.11	0.07	0.18	0.18	0.10	0.09	0.55	0.84	0.83	0.93
	3	**0.01**	**0.01**	**0.00**	**0.00**	**0.01**	**0.05**	**0.01**	**0.01**	**0.03**	0.16	0.12	0.26
	4	0.73	0.45	0.18	0.19	0.33	0.35	0.34	0.37	0.17	0.45	0.63	0.73
	all	0.10	0.19	0.32	0.52	**0.00**	**0.00**	**0.00**	**0.00**	0.76	1.00	1.00	1.00

TABLE IV
FEATURE IMPORTANCE.EACH CELL CONTAINS THE RANKING OF THE $sim2r_M$ FEATURE WHEN SORTING THE 16 FEATURES IN DESCENDING ORDER OF THEIR AVERAGE FEATURE IMPORTANCE IN THE DP MODELS.

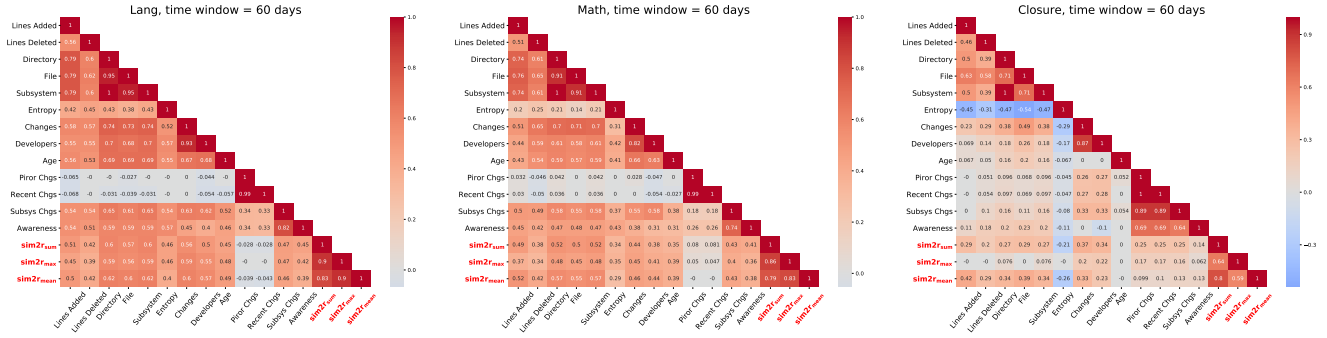
		$sim2r_{sum}$				$sim2r_{mean}$				$sim2r_{max}$			
project	fold	30	60	90	120	30	60	90	120	30	60	90	120
Lang	all	8	**4**	**2**	**2**	10	**3**	5	6	11	8	**4**	**3**
Math	all	8	7	**4**	**4**	11	**4**	**2**	**2**	12	10	8	8
Closure	all	**4**	**4**	**4**	**4**	9	7	5	5	6	5	6	7

IV. RESULT AND ANALYSIS

This section presents the results of our empirical evaluation and responds to the research questions.

A. RQ1. Feature Analysis

Figure 2 presents the heatmaps of Spearman correlation coefficients between the 16 features used to train our defect prediction models. Overall, for all three projects, our three new features, $sim2r_{sum}$, $sim2r_{max}$, and $sim2r_{mean}$, are weakly correlated to the existing ones (lighter shade), while being strongly correlated to each other (darker shade). From these, we argue that $sim2r_M$ features do contain some extra information that is not captured by the existing features. While the figures show only the results of using a time window of 60 days, the other three windows also share the similar trend.



(a) Lang (b) Math (c) Closure

Fig. 2. Heat maps of Spearman correlation coefficients between the 16 features. The coefficients whose p-values are greater than 0.05 are replaced with zero.

Table III shows the results of Mann-Whitney U-test between $sim2r_M$ feature values from fix-inducing and non-inducing commits. We have performed a two-tailed test with the alternative hypothesis that fix-inducing commits have higher $sim2r_M$ values than non-inducing commits. Each cell in the table contains the p-value from a specific U-test. Among three $sim2r_M$ features, $sim2r_{max}$ shows the weakest correlation with its p-values rarely below 0.05, failing to reject the null-hypothesis. $sim2r_{mean}$ shows the strongest correlation, rejecting the null-hypothesis for all three projects; $sim2r_{sum}$ sits in the middle. We suspect that these different correlations may stem from how these features are aggregated. $sim2r_{max}$ takes the maximum similarity between bug reports and code changes; it is more easily affected by any *outlier* bug report coincidently similar to the commit under consideration than the other features. In contrast, $sim2r_{mean}$ is the most stable due to taking the arithmetic mean; $sim2r_{sum}$ is less sensitive to outliers than $sim2r_{max}$, but susceptible to the number of bug reports as it adds their similarities to the commit.

Table IV shows the rankings of the $sim2r_M$ features when the 16 features used to train DP models are sorted in descending order of their feature importance values in those models: if a feature is within the top four most important features (25%), its ranking is written in bold. Overall, at least one variation of the $sim2r_M$ features is in the top four, further emphasising the usefulness of $sim2r_M$ features.

Answer to RQ1: $sim2r_M$ features capture distinct and useful properties of fix-inducing commits. Between three $sim2r_M$ features, $sim2r_{mean}$ is the most useful one, followed by $sim2r_{sum}$, and then $sim2r_{max}$.

B. RQ2. Effectiveness

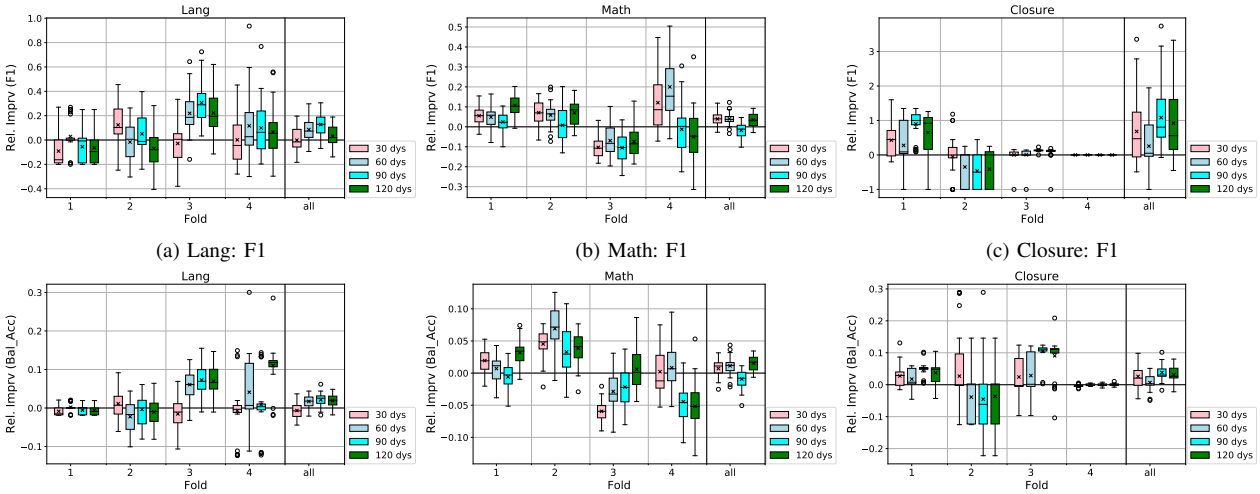
Tables V to VII compare the performance of defect prediction with (denoted by *IRFL*) and without using $sim2r_M$ features (denoted by *base*). Any performance metric value in bold text indicates that the use of $sim2r_M$ features outperforms the baseline; the values in percentage denote the degree of relative improvement. The results are evaluated both for each fold (*fold* 1 to 4) and for the union of these four folds (*all*). The row *all* in Tables V to VII, and the rightmost plots of Figures 3a to 3f, show the summary of the results: overall,

TABLE V
THE EFFECTIVENESS OF $sim2r_M$ FEATURES IN LANG. THE VALUES IN PERCENTAGE DENOTE THE RELATIVE IMPROVEMENT FROM USING THE $sim2r_M$ FEATURES. THE UNDERLINED VALUE MEANS THAT THE RESPECTIVE IMPROVEMENT IS STATISTICALLY SIGNIFICANT ($p \leq 0.05$) AND HAS AT LEAST A MEDIUM EFFECT SIZE ($\delta \geq 0.33$).

F	metric	base	IRFL: with $sim2r_M$ features 30 days		60 days		90 days		120 days	
1	AUC	0.698	0.671	-3.9%	0.658	-5.7%	0.662	-5.2%	0.681	-2.4%
	Acc	0.929	0.929	0.0%	0.929	0.0%	0.928	-0.1%	0.928	-0.1%
	Acc_{bal}	0.540	0.535	-0.9%	**0.541**	**0.2%**	0.537	-0.6%	0.537	-0.6%
	F1	0.146	0.132	-9.6%	**0.149**	**2.1%**	0.137	-6.2%	0.135	-7.5%
	PR	0.195	**0.203**	**4.1%**	**0.204**	**4.6%**	**0.205**	**5.1%**	**0.201**	**3.1%**
2	AUC	0.799	**0.829**	**3.8%**	**0.842**	**5.4%**	**0.829**	**3.8%**	**0.833**	**4.3%**
	Acc	0.932	**0.940**	**0.9%**	**0.937**	**0.5%**	**0.938**	**0.6%**	0.926	-0.6%
	Acc_{bal}	0.647	**0.654**	**1.1%**	0.632	-2.3%	0.645	-0.3%	0.640	-1.1%
	F1	0.259	**0.286**	**10.4%**	0.252	-2.7%	**0.268**	**3.5%**	0.237	-8.5%
	PR	0.189	**0.230**	**21.7%**	**0.202**	**6.9%**	0.183	-3.2%	0.176	-6.9%
3	AUC	0.682	**0.692**	**1.5%**	**0.729**	**6.9%**	**0.712**	**4.4%**	**0.717**	**5.1%**
	Acc	0.900	**0.907**	**0.8%**	0.895	-0.6%	**0.902**	**0.2%**	0.890	-1.1%
	Acc_{bal}	0.592	0.582	-1.7%	**0.627**	**5.9%**	**0.634**	**7.1%**	**0.633**	**6.9%**
	F1	0.195	0.187	-4.1%	**0.234**	**20.0%**	**0.251**	**28.7%**	**0.234**	**20.0%**
	PR	0.163	**0.167**	**2.5%**	**0.187**	**14.7%**	**0.192**	**17.8%**	**0.186**	**14.1%**
4	AUC	0.866	**0.874**	**0.9%**	**0.886**	**2.3%**	**0.871**	**0.6%**	**0.876**	**1.2%**
	Acc	0.925	**0.926**	**0.1%**	**0.926**	**0.1%**	**0.931**	**0.6%**	0.906	-2.1%
	Acc_{bal}	0.757	0.751	-0.8%	**0.787**	**4.0%**	**0.763**	**0.8%**	**0.840**	**11.0%**
	F1	0.092	0.090	-2.2%	**0.101**	**9.8%**	**0.100**	**8.7%**	**0.096**	**4.3%**
	PR	0.058	0.055	-5.2%	**0.063**	**8.6%**	**0.067**	**15.5%**	**0.090**	**55.2%**
all	AUC	0.644	**0.649**	**0.8%**	**0.684**	**6.2%**	**0.666**	**3.4%**	**0.663**	**3.0%**
	Acc	0.921	**0.926**	**0.5%**	**0.922**	**0.1%**	**0.925**	**0.4%**	0.912	-1.0%
	Acc_{bal}	0.584	0.580	-0.7%	**0.593**	**1.5%**	**0.596**	**2.1%**	**0.595**	**1.9%**
	F1	0.177	0.176	-0.6%	**0.191**	**7.9%**	**0.199**	**12.4%**	**0.182**	**2.8%**
	PR	0.110	**0.113**	**2.7%**	**0.124**	**12.7%**	**0.125**	**13.6%**	**0.123**	**11.8%**

DP models using $sim2r_M$ features can often outperform those that do not. The underlined values in Tables V to VII show that the improvement of the evaluation metric computed from the 30 repetitions is statistically significant ($p \leq 0.05$) with a moderate effect size ($\delta > 0.33$) in most cases.

As described in Figure 3, the impact of using $sim2r_M$ features varies depending on the size of the time window that was used to collect bug reports. Overall, there does not exist the *best* value for the time window that outperforms all the others, particularly since this value also changes during the software development. For example, in Lang, 30 days is the best choice for the time window in fold two, whereas, in fold three, 90 days is the better choice. To maximise the effectiveness of $sim2r_M$ features, the time window should be tuned for each project and adjusted dynamically throughout the software development. Nevertheless, within these results, the best value for the time window would be 90 days for



(a) Lang: F1 (b) Math: F1 (c) Closure: F1

(d) Lang: Balanced Accuracy (e) Math: Balanced Accuracy (f) Closure: Balanced Accuracy

Fig. 3. Relative Improvements in F1 score and Balanced Accuracy: while the degree of improvement varies depending on both time window and fold, using $sim2r_M$ features often improves the overall performance of defective prediction models ($Rel.Imprv_{all} > 0$)

TABLE VI
THE EFFECTIVENESS OF $sim2r_M$ FEATURES IN MATH.

F	metric	base	IRFL: with $sim2r_M$ features 30 days		60 days		90 days		120 days	
1	AUC	0.787	**0.797**	**1.3%**	**0.790**	**0.4%**	**0.794**	**0.9%**	**0.799**	**1.5%**
	Acc	0.826	**0.834**	**1.0%**	**0.847**	**2.5%**	**0.852**	**3.1%**	**0.847**	**2.5%**
	Acc_{bal}	0.656	**0.668**	**1.8%**	**0.660**	**0.6%**	0.652	-0.6%	**0.677**	**3.2%**
	F1	0.414	**0.436**	**5.3%**	**0.433**	**4.6%**	**0.424**	**2.4%**	**0.457**	**10.4%**
	PR	0.429	**0.447**	**4.2%**	**0.444**	**3.5%**	**0.442**	**3.0%**	**0.459**	**7.0%**
2	AUC	0.833	**0.844**	**1.3%**	**0.849**	**1.9%**	**0.842**	**1.1%**	**0.842**	**1.1%**
	Acc	0.896	0.894	-0.2%	0.882	-1.6%	0.882	-1.6%	0.896	0.0%
	Acc_{bal}	0.714	**0.746**	**4.5%**	**0.763**	**6.9%**	**0.737**	**3.2%**	**0.741**	**3.8%**
	F1	0.430	**0.459**	**6.7%**	**0.454**	**5.6%**	**0.432**	**0.5%**	**0.459**	**6.7%**
	PR	0.475	**0.490**	**3.2%**	0.459	-3.4%	0.449	-5.5%	0.464	-2.3%
3	AUC	0.763	0.763	0.0%	0.759	-0.5%	**0.766**	**0.4%**	0.758	-0.7%
	Acc	0.866	**0.876**	**1.2%**	0.863	-0.3%	0.844	-2.5%	0.835	-3.6%
	Acc_{bal}	0.682	0.641	-6.0%	0.662	-2.9%	0.667	-2.2%	**0.685**	**0.4%**
	F1	0.336	0.301	-10.4%	0.312	-7.1%	0.300	-10.7%	0.310	-7.7%
	PR	0.221	**0.226**	**2.3%**	0.203	-8.1%	0.197	-10.9%	0.203	-8.1%
4	AUC	0.833	**0.840**	**0.8%**	**0.841**	**1.0%**	**0.845**	**1.4%**	**0.845**	**1.4%**
	Acc	0.944	**0.953**	**1.0%**	**0.957**	**1.4%**	**0.955**	**1.2%**	**0.955**	**1.2%**
	Acc_{bal}	0.643	**0.644**	**0.2%**	**0.648**	**0.8%**	0.614	-4.5%	0.610	-5.1%
	F1	0.303	**0.337**	**11.2%**	**0.360**	**18.8%**	0.297	-2.0%	0.285	-5.9%
	PR	0.283	**0.311**	**9.9%**	**0.322**	**13.8%**	**0.297**	**4.9%**	**0.288**	**1.8%**
all	AUC	0.809	**0.817**	**1.0%**	0.809	0.0%	0.806	-0.4%	0.809	0.0%
	Acc	0.883	**0.889**	**0.7%**	**0.887**	**0.5%**	0.883	0.0%	0.883	0.0%
	Acc_{bal}	0.679	**0.684**	**0.7%**	**0.687**	**1.2%**	0.673	-0.9%	**0.690**	**1.6%**
	F1	0.385	**0.400**	**3.9%**	**0.401**	**4.2%**	0.378	-1.8%	**0.398**	**3.4%**
	PR	0.344	**0.365**	**6.1%**	**0.347**	**0.9%**	0.329	-4.4%	0.336	-2.3%

TABLE VII
THE EFFECTIVENESS OF $sim2r_M$ FEATURES IN CLOSURE.

F	metric	base	IRFL: with $sim2r_M$ features 30 days		60 days		90 days		120 days	
1	AUC	0.671	0.669	-0.3%	0.644	-4.0%	0.626	-6.7%	0.635	-5.4%
	Acc	0.941	0.936	-0.5%	**0.946**	**0.5%**	**0.949**	**0.9%**	**0.948**	**0.7%**
	Acc_{bal}	0.519	**0.533**	**2.7%**	**0.528**	**1.7%**	**0.544**	**4.8%**	**0.538**	**3.7%**
	F1	0.078	**0.120**	**53.8%**	**0.105**	**34.6%**	**0.161**	**106.4%**	**0.140**	**79.5%**
	PR	0.127	**0.140**	**10.2%**	**0.154**	**21.3%**	**0.181**	**42.5%**	**0.189**	**48.8%**
2	AUC	0.815	**0.833**	**2.2%**	0.798	-2.1%	0.785	-3.7%	0.808	-0.9%
	Acc	0.982	**0.984**	**0.2%**	**0.983**	**0.1%**	**0.983**	**0.1%**	**0.983**	**0.1%**
	Acc_{bal}	0.547	**0.559**	**2.2%**	0.524	-4.2%	0.519	-5.1%	0.524	-4.2%
	F1	0.156	**0.189**	**21.2%**	0.083	-46.8%	0.065	-58.3%	0.083	-46.8%
	PR	0.326	**0.417**	**27.9%**	**0.452**	**38.7%**	**0.427**	**31.0%**	**0.474**	**45.4%**
3	AUC	0.684	0.679	-0.7%	0.666	-2.6%	0.649	-5.1%	0.653	-4.5%
	Acc	0.964	0.962	-0.2%	0.963	-0.1%	**0.966**	**0.2%**	0.962	-0.2%
	Acc_{bal}	0.495	**0.507**	**2.4%**	**0.509**	**2.8%**	**0.547**	**10.5%**	**0.540**	**9.1%**
	F1	0.004	**0.034**	**750.0%**	**0.038**	**850.0%**	**0.128**	**3100.0%**	**0.106**	**2550.0%**
	PR	0.073	**0.088**	**20.5%**	**0.088**	**20.5%**	**0.116**	**58.9%**	**0.102**	**39.7%**
4	AUC	0.843	0.807	-4.3%	0.791	-6.2%	0.843	0.0%	0.830	-1.5%
	Acc	0.988	0.984	-0.4%	0.988	0.0%	0.987	-0.1%	0.988	0.0%
	Acc_{bal}	0.496	0.495	-0.2%	0.496	0.0%	0.496	0.0%	**0.497**	**0.2%**
	F1	0.000	0.000	0.0%	0.000	0.0%	0.000	0.0%	0.000	0.0%
	PR	0.025	0.022	-12.0%	0.021	-16.0%	**0.026**	**4.0%**	0.024	-4.0%
all	AUC	0.745	**0.750**	**0.7%**	0.726	-2.6%	0.710	-4.7%	0.724	-2.8%
	Acc	0.969	0.966	-0.3%	**0.970**	**0.1%**	**0.971**	**0.2%**	**0.970**	**0.1%**
	Acc_{bal}	0.518	**0.531**	**2.5%**	**0.521**	**0.6%**	**0.537**	**3.7%**	**0.533**	**2.9%**
	F1	0.064	**0.096**	**50.0%**	**0.074**	**15.6%**	**0.123**	**92.2%**	**0.110**	**71.9%**
	PR	0.082	**0.096**	**17.1%**	**0.095**	**15.9%**	**0.106**	**29.3%**	**0.106**	**29.3%**

Lang and Closure and 30 or 60 days for Math. With these time windows, $sim2r_M$ features improve F1 score by 12.4%, 4.2%, and 92.2% and Balanced Accuracy by 2.1%, 1.2%, and 3.7%, respectively for Lang, Math, and Closure.

We observe that $sim2r_M$ features mainly improve F1 score and PR-AUC while increasing the other metrics evaluated over the entire set, i.e., AUC, Accuracy, and Balanced Accuracy, by a small degree, at least when using the best time window. For example, in Lang, $sim2r_M$ features with the 90 days window improve F1 and PR-AUC by 12.4% and 13.6%, respectively; for the other metrics, the relative improvements are 3.4%, 0.4%, and 2.1%, respectively for AUC, Accuracy, and Balanced Accuracy. This increase in F1 score and PR-AUC along with the small increase in the other metrics implies that fix-inducing commits that were previously predicted as non-inducing are correctly predicted as fix-inducing by using $sim2r_M$ features, reducing the number of false-negatives, which in our case, are fix-inducing commits predicted as non-inducing. Bug reports explain bugs that were induced by fix-inducing commits. We argue that by directly comparing code changes to bug reports, $sim2r_M$ features capture the properties of fix-inducing commits that are related to the context of induced bugs, which the previous features based on the meta-data of fix-inducing commits were unable to capture.

Figure 3 shows that the degree of improvement varies depending on folds. While there is no strong general trend, the improvement is more notable in folds with more fix-inducing commits, such as the first fold of Math and Closure.

Answer to RQ2: $sim2r_M$ features can improve the performance of defect prediction models, given sufficient fix-

inducing commits to train and test. Benefits of $sim2r_M$ can be further maximised by tuning the time window for each project.

C. RQ3. Actionability

Table VIII and Figure 4 compare the efforts spent until encountering the first defective file between when following the order of $susp_f$, and when following the baselines, which are random ordering, ordered by closed bug count (cnt_{CB}), and ordered by closed bug similarity (sim_{CB}). To focus on the cases where developers can save their effort by knowing defective files in advance, we consider only the fix-inducing commits that also contain non-defective changes, for evaluation. As there are always more than one modified files in these cases, the expected rank of the random approach is always greater than one, resulting in $acc@1$ to be zero. Compared to the random ordering, the use of $susp_f$ successfully increases $acc@1$ of Lang, Math, and Closure from zero up to 42, 93, and 28, respectively. While the degree of increase in $acc@n$ decreases as n grows, developers can still find the defective file earlier using $susp_f$, when compared to random inspection.

The effectiveness of $susp_f$ tends to increase as the time window increases. For example, as the time window changes from 30 to 60 days, $acc@1$ increases from 70 to 90 in Math. However, an unnecessarily large time window may also harm the effectiveness of $susp_f$, since it becomes more likely that the used bug reports are now outdated: $acc@1$ decreases from 93 to 89 in Math when the time window changes from 90 to 120 days. Similarly to our answer to RQ2, we may need to tune the time window for each project to maximise the benefit of $susp_f$. In general, Table VIII shows that a time window of 90 to 120 days achieves fair performance.

Inspecting files following the order of $susp_f$ can easily outperform the other two baselines, cnt_{CB} and sim_{CB}. For all three projects, $acc@n$ with $susp_f$ is significantly higher: for Lang and Math, $acc@1$ increases at least by 2600% and 1067%, and by 800% and 592%, when compared to cnt_{CB} and sim_{CB}, respectively. For Closure, while both cnt_{CB} and sim_{CB} failed to rank any defective file at the top for any fix-inducing commits ($acc@1 = 0$), $susp_f$ successfully does so for at lest 27 commits ($acc@1 = 27$). Unlike the random approach, cnt_{CB} and sim_{CB} do take past faults into account, similarly to $susp_f$. We argue that the difference in actionability comes from whether an approach takes files that were non-defective in the past into account: $susp_f$ does by using past suspiciousness of files regardless of whether they were defective in the past, whereas cnt_{CB} and sim_{CB} do not by considering only the files that were defective in the past. Consequently, from the improvement gained by using $susp_f$ over cnt_{CB} and sim_{CB}, we suspect that the previously suspicious files that were not actually defective may turn out to be relevant to defective changes in subsequent commits that we investigated.

Figure 4 visualises the complete comparison between $susp_f$ and the baselines. Regardless of the choice of the time window, $susp_f$ always reduces the inspection effort of developers by successfully ranking defective files near the top.

Table IX presents how the effectiveness of $susp_f$ changes as more reports are included for computation of $susp_f$. Here, CB denotes closed bug repots, B denotes bug reports only with the status of open or close, and A denotes all types of bug reports. In Lang and Math, $acc@n$ increases as more report types are considered. For example, by switching from using only CB to A, the degree of improvement in $acc@1$ increases from at least 592% to at least 908% for Math with 60 days window. While $acc@n$ decreases for Closure by considering open bug reports in addition (i.e., going from C to B), $susp_f$ is the most effective when all types of bug reports are taken into consideration (i.e., A). These results suggest that we can further improve actionability of DP results by leveraging more bug reports. Table X reports how much effort (measured in the number of files inspected) we can save by employing $susp_f$ computed using all types of bug reports within the time window. On average, developers can inspect two to nine fewer files before meeting the first defective file.

Answer to RQ3: $susp_f$ can make DP results more actionable by successfully guiding developers to defective files based on the past suspiciousness. Comparison to the baselines suggests that the previously suspicious, but non-defective files can still be relevant to defective changes in the future commits.

V. Threats to Validity

The primary threat to validity of this study is whether the data collection and model inference have been performed correctly. To mitigate this threat, we use the SZZ algorithm [23] that is widely used for defect prediction research [3], [13]. We use the open source implementation [24], with additional filters from previous work [3] that are designed to reduce false-positives. For model inference, we use the widely used machine learning package, `scikit-learn` [32].

Threats to external validity concerns any factors that affects how generalisable the results of this study are. The biggest factor is the fact that our results are based on Defects4J, a benchmark of real-world faults [21]. Defects4J has been extensively studied in the context of Fault Localisation (FL), and we use it to exploit the synergy between DP and FL. However, further studies are needed to confirm our results for the benchmark datasets in the area of defect prediction.

Threats to construct validity concerns whether our measurement actually reflects what we intend to measure. All our evaluation metrics are standard metrics used for binary classification, which is how defect prediction is formulated.

VI. Related Work

Our aim is to improve both accuracy and actionability of defect prediction results by considering previous faults via fault localisation techniques. Historical faults have been studied in relation to recent or future faults. Yu et al. generated a mathematical model that predicts the probability of a subsystem having a defect based on the correlation between the number of past and future faults [33]. Similarly, Graves et al. utilised the number of past defects to generate a model for predicting the number of future faults in a program module [34]. Exploiting

TABLE VIII

ACTIONABILITY ($acc@n$): COLUMN n DENOTES THE n IN $acc@n$. THE PARENTHESISED NUMBER IN COLUMN PROJECT IS THE TOTAL NUMBER OF EVALUATED FIX-INDUCING COMMITS. THE OTHER PARENTHESISED VALUES ARE THE RELATIVE IMPROVEMENTS. FOR cnt_{CB} AND sim_{CB}, WE COMPARED THEM WITH THE RANDOM (R). FOR $susp_f$, THESE VALUSE ARE ARE THE RELATIVE IMPROVEMENTS OVER R, cnt_{CB}, AND sim_{CB}.

Project	n	R	30 days			60 days			90 days			120 days		
			cnt_{CB}	sim_{CB}	$susp_f$	cnt_{CB}	sim_{CB}	$susp_f$	cnt_{CB}	sim_{CB}	$susp_f$	cnt_{CB}	sim_{CB}	$susp_f$
Lang (131)	1	0	1 (—)	2 (—)	**27** (— ,26.00,12.50)	1 (—)	3 (—)	**33** (— ,32.00,10.00)	1 (—)	4 (—)	**36** (— ,35.00, 8.00)	1 (—)	4 (—)	**42** (— ,41.00, 9.50)
	3	78	79 (0.01)	79 (0.01)	**94** (0.21, 0.19, 0.19)	81 (0.04)	81 (0.04)	**93** (0.19, 0.15, 0.15)	82 (0.05)	82 (0.05)	**96** (0.23, 0.17, 0.17)	82 (0.05)	82 (0.05)	**98** (0.26, 0.20, 0.20)
	5	98	99 (0.01)	99 (0.01)	**109** (0.11, 0.10, 0.10)	101 (0.03)	101 (0.03)	**108** (0.10, 0.07, 0.07)	102 (0.04)	102 (0.04)	**111** (0.13, 0.09, 0.09)	102 (0.04)	102 (0.04)	**110** (0.12, 0.08, 0.08)
Math (324)	1	0	6 (—)	10 (—)	**70** (— ,10.67, 6.00)	4 (—)	13 (—)	**90** (— ,21.50, 5.92)	4 (—)	12 (—)	**93** (— ,22.25, 6.75)	4 (—)	12 (—)	**89** (— ,21.25, 6.42)
	3	151	155 (0.03)	155 (0.03)	**201** (0.33, 0.30, 0.30)	152 (0.01)	153 (0.01)	**208** (0.38, 0.37, 0.36)	151 (0.00)	152 (0.01)	**207** (0.37, 0.37, 0.36)	151 (0.00)	152 (0.01)	**204** (0.35, 0.35, 0.34)
	5	199	205 (0.03)	205 (0.03)	**230** (0.16, 0.12, 0.12)	205 (0.03)	205 (0.03)	**238** (0.20, 0.16, 0.16)	204 (0.03)	204 (0.03)	**239** (0.20, 0.17, 0.17)	204 (0.03)	204 (0.03)	**236** (0.19, 0.16, 0.16)
Closure (77)	1	0	0 (—)	0 (—)	**28** (— , — , —)	0 (—)	0 (—)	**27** (— , — , —)	0 (—)	0 (—)	**28** (— , — , —)	0 (—)	0 (—)	**27** (— , — , —)
	3	38	38 (0.00)	38 (0.00)	**55** (0.45, 0.45, 0.45)	38 (0.00)	38 (0.00)	**56** (0.47, 0.47, 0.47)	38 (0.00)	38 (0.00)	**59** (0.55, 0.55, 0.55)	38 (0.00)	38 (0.00)	**58** (0.53, 0.53, 0.53)
	5	51	51 (0.00)	51 (0.00)	**66** (0.29, 0.29, 0.29)	51 (0.00)	51 (0.00)	**67** (0.31, 0.31, 0.31)	51 (0.00)	51 (0.00)	**67** (0.31, 0.31, 0.31)	51 (0.00)	51 (0.00)	**66** (0.29, 0.29, 0.29)

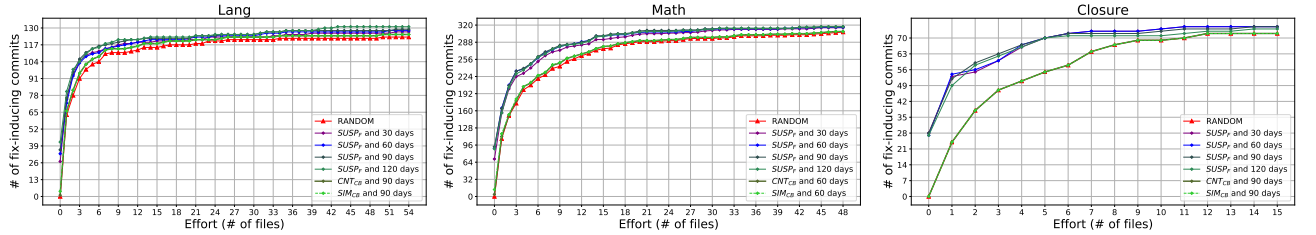


Fig. 4. The effort spent before meeting the first defective file: the x-axis is the number of non-defective files that are inspected, and the y-axis represents the number of fix-inducing commits in which at least one of their defective file is located in the top $x + 1$. For better visualisation, the graphs cover up to only 95% of the fix-inducing commits. For cnt_{CB} and sim_{CB}, we report only the best results.

,

TABLE IX

COMPARISON BETWEEN THREE TYPES OF $susp_f$ USING THREE DIFFERENT SETS OF REPORTS (CB, B, AND A). THE ONE WITH THE BEST PERFORMANCE IS IN BOLD TEXT.

Project	n	30 days			60 days			90 days			120 days		
		CB	B	A	CB	B	A	CB	B	A	CB	B	A
Lang (131)	1	27	**49**	**49**	33	**51**	**51**	36	**53**	**53**	42	**57**	**57**
	3	94	**101**	**101**	93	**99**	**99**	**96**	**96**	**96**	98	**99**	**99**
	5	109	**111**	**111**	108	**111**	**111**	111	**113**	**113**	110	**111**	**111**
Math (324)	1	70	88	**123**	90	94	**131**	93	100	**134**	89	95	**125**
	3	201	206	**233**	208	204	**243**	207	204	**238**	204	206	**237**
	5	230	238	**261**	238	242	**266**	239	247	**264**	236	244	**263**
Closure (77)	1	28	26	**32**	27	20	**30**	28	17	**31**	27	18	**32**
	3	55	52	**60**	56	50	**58**	59	49	**60**	58	48	**60**
	5	66	65	**69**	**67**	61	**67**	**67**	64	65	**66**	63	65

TABLE X

EFFORT SAVED FROM USING $susp_f$ WITH ALL REPORTS (A) (AVG. # OF FILES): *cnt* AND *sim* ARE THE SHORT OF CNT_{CB} AND SIM_{CB}.

Project	30 days			60 days			90 days			120 days		
	R	cnt	sim	R	cnt	sim	R	cnt	sim	R	cnt	sim
Lang	4.5	4.5	4.5	4.7	3.9	4.0	4.7	3.3	3.3	5.1	3.8	3.8
Math	8.1	4.9	4.9	8.8	5.6	5.5	9.0	5.9	5.8	8.6	5.5	5.5
Closure	2.4	2.4	2.4	2.1	2.1	2.1	2.0	2.0	2.0	1.9	1.9	1.9

the correlation even further, Hassan and Holt generated the list of the top ten subsystems that are most susceptible to have faults based on their previous faults [35]. Zimmerman et al. also showed that the number of pre-release and post-release faults are correlated [36]. While all these studies highlight the correlation between past and future faults, most of them stop at looking at the number and location of faults. We try to provide richer information for defect prediction by adopting a fault localisation technique.

Recently, Bowes et al. proposed a new defect prediction technique that uses the results of mutation testing on a target program [25]. The results showed that the performance of defect prediction could be improved by using the dynamic analysis. While this work is similar in the sense that it exploits information from other faults, it uses artificial mutants whereas we directly leverage previous faults via fault localisation.

VII. CONCLUSION

We focus on the observation that while both fault localisation and defect prediction try to assure a certain degree of software quality, the impact each process has on the other has been rarely investigated. As the first step of exploring this mostly untouched area from the side of defect prediction, we leverage the past suspiciousness scores of code changes computed from fault localisation to improve defect prediction. The experimental results on three open-source projects confirm that the past suspiciousness of code can enhance defect prediction by improving the performance of JIT defect prediction models and assisting developers on processing fix-inducing commits via further directing them to defective files.

ACKNOWLEDGEMENT

Jeongju Sohn and Shin Yoo were supported by National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (Grant No. NRF-2020R1A2C1013629). Yasutaka Kamei was partially supported by JSPS KAKENHI Grant Numbers JP18H03222 and JSPS International Joint Research Program with SNSF (Project "SENSOR").

REFERENCES

[1] C. Catal, "Software fault prediction: A literature review and current trends," *Expert Systems with Applications*, vol. 38, no. 4, pp. 4626 – 4636, 2011.

[2] Y. Kamei and E. Shihab, "Defect prediction: Accomplishments and future challenges," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5, 2016, pp. 33–45.

[3] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 412–428, May 2018.

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

[5] T.-D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: Better together," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 579–590.

[6] 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.

[7] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 262–273. [Online]. Available: https://doi.org/10.1145/2970276.2970359

[8] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localisation," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA 2017, 2017, pp. 273–283.

[9] J. Sohn and S. Yoo, "Empirical evaluation of fault localisation using code and change metrics," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[10] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting faults from cached history," in *29th International Conference on Software Engineering (ICSE'07)*, May 2007, pp. 489–498.

[11] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu, "Bugcache for inspections: Hit or miss?" in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 322–331. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025157

[12] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[13] 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*, ser. FSE '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2393596.2393670

[14] M. Papadakis and Y. L. Traon, "Metallaxis-fl: mutation-based fault localization," *Softw. Test., Verif. Reliab.*, vol. 25, no. 5-7, pp. 605–628, 2015. [Online]. Available: http://dx.doi.org/10.1002/stvr.1509

[15] X. Li, S. Zhu, M. d'Amorim, and A. Orso, "Enlightened debugging," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 82–92. [Online]. Available: https://doi.org/10.1145/3180155.3180242

[16] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman, "Human competitiveness of genetic programming in sbfl: Theoretical and empirical analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 26, no. 1, pp. 4:1–4:30, July 2017.

[17] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 169–180. [Online]. Available: https://doi.org/10.1145/3293882.3330574

[18] G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Commun. ACM*, vol. 18, no. 11, pp. 613–620, Nov. 1975.

[19] J. Romano, J. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys?" in *annual meeting of the Florida Association of Institutional Research*, 2006, pp. 1–3.

[20] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead, "Does bug prediction support human developers? findings from a google case study," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 372–381.

[21] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440.

[22] Atlassian. [Online]. Available: https://www.atlassian.com/software/jira

[23] A. Z. Jacek Śliwerski, Thomas Zimmermann, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, ser. MSR '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 1–5. [Online]. Available: https://doi.org/10.1145/1083142.1083147

[24] M. Borg, O. Svensson, K. Berg, and D. Hansson, "Szz unleashed: An open implementation of the szz algorithm - featuring example usage in a study of just-in-time bug prediction for the jenkins project," in *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, ser. MaLTeSQuE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 7–12. [Online]. Available: https://doi.org/10.1145/3340482.3342742

[25] D. Bowes, T. Hall, M. Harman, Y. Jia, F. Sarro, and F. Wu, "Mutation-aware fault prediction," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 330–341. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931039

[26] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, "Heterogeneous defect prediction," *IEEE Transactions on Software Engineering*, vol. 44, no. 9, pp. 874–896, 2018.

[27] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, 2008.

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

[29] S. Tabassum, L. L. Minku, D. Feng, G. G. Cabral, and L. Song, "An investigation of cross-project learning in online just-in-time software defect prediction," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 554–565. [Online]. Available: https://doi.org/10.1145/3377811.3380403

[30] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, "Heterogeneous defect prediction," *IEEE Transactions on Software Engineering*, vol. PP, pp. 1–1, 06 2017.

[31] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[33] T. . Yu, V. Y. Shen, and H. E. Dunsmore, "An analysis of several software defect models," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1261–1270, 1988.

[34] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[35] A. E. Hassan and R. C. Holt, "The top ten list: dynamic fault prediction," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 263–272.

[36] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, 2007, pp. 9–9.