Mining Action Rules for Defect Reduction Planning

KHOULOUD OUESLATI, Polytechnique Montréal, Canada
GABRIEL LABERGE, Polytechnique Montréal, Canada
MAXIME LAMOTHE, Polytechnique Montréal, Canada
FOUTSE KHOMH, Polytechnique Montréal, Canada

Defect reduction planning plays a vital role in enhancing software quality and minimizing software maintenance costs. By training a black box machine learning model and "explaining" its predictions, explainable AI for software engineering aims to identify the code characteristics that impact maintenance risks. However, post-hoc explanations do not always faithfully reflect what the original model computes. In this paper, we introduce CounterACT, a Counterfactual ACTion rule mining approach that can generate defect reduction plans without black-box models. By leveraging action rules, CounterACT provides a course of action that can be considered as a counterfactual explanation for the class (e.g., buggy or not buggy) assigned to a piece of code. We compare the effectiveness of CounterACT with the original action rule mining algorithm and six established defect reduction approaches on 9 software projects. Our evaluation is based on (a) overlap scores between proposed code changes and actual developer modifications; (b) improvement scores in future releases; and (c) the precision, recall, and F1-score of the plans. Our results show that, compared to competing approaches, CounterACT's explainable plans achieve higher overlap scores at the release level (median 95%) and commit level (median 85.97%), and they offer better trade-off between precision and recall (median F1-score 88.12%). Finally, we venture beyond planning and explore leveraging Large Language models (LLM) for generating code edits from our generated plans. Our results show that suggested LLM code edits supported by our plans are actionable and are more likely to pass relevant test cases than vanilla LLM code recommendations.

CCS Concepts: • **Software and its engineering** → **Maintaining software**; • **Computing methodologies** → **Planning under uncertainty**; **Rule learning**.

Additional Key Words and Phrases: Software analytics, Defect reduction planning, Explainability, Action rule mining, Counterfactual explanations

ACM Reference Format:
Khouloud Oueslati, Gabriel Laberge, Maxime Lamothe, and Foutse Khomh. 2024. Mining Action Rules for Defect Reduction Planning. *Proc. ACM Softw. Eng.* 1, FSE, Article 102 (July 2024), 23 pages. https://doi.org/10.1145/3660809

1 INTRODUCTION

In today's software-driven society, Software Quality Assurance (SQA) has become a crucial practice to ensure that software products meet the required standards of functionality, reliability, usability, and performance. Since fixing defects is costly [12], addressing them before the software is released can typically result in faster and cheaper remediation [4], yet it remains a critical challenge to identify and fix defects early in the development lifecycle. Over the past decade, numerous Software

Authors' Contact Information: Khouloud Oueslati, Polytechnique Montréal, Montreal, Canada, KHOULOUD.OUESLATI@POLYMTL.CA; Gabriel Laberge, Polytechnique Montréal, Montreal, Canada, gabriel.laberge@polymtl.ca; Maxime Lamothe, Polytechnique Montréal, Montreal, Canada, maxime.lamothe@polymtl.ca; Foutse Khomh, Polytechnique Montréal, Montreal, Canada, foutse.khomh@polymtl.ca.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2024 Copyright held by the owner/author(s).
ACM 2994-970X/2024/7-ART102
https://doi.org/10.1145/3660809

Defect Prediction (SDP) techniques emerged in order to predict the high-risk areas of source code that are prone to post-release defects [9, 34, 39, 56, 62]. However, one of the significant limitations faced by SDP models is their lack of explanation and actionability. Due to the absence of practical recommendations that developers can act upon, practitioners might be skeptical of predictions that seem counterintuitive which may hinder the adoption of software analytics in practice [8]. Indeed, a prior survey investigating practitioners' perceptions of SQA rule-based planning [48], found that 52%-80% of respondents perceived it as useful, and 52%-72% of the respondents expressed willingness to adopt rule-based guidance. Furthermore, the release of commercial AI-driven defect prediction tools such as Amazon's CodeGuru, and Microsoft's Code Defect AI, implies practitioners' interest in defect reduction planning. The latter even uses explainable Just-In-Time defect prediction to provide a visualization of the software metrics that contribute to general code improvement guidelines. While several research studies have leveraged model-agnostic post-hoc explainability techniques such as LIME [17] and TimeLIME [45], to understand file-level defect predictions of black-box ML models, post-hoc explanations are not always trustworthy. They can be misleading and do not always faithfully reflect what the original model computes [52]. Moreover, previous work [56] has shown that they can be inconsistent and unreliable under different settings [57]. Consequently, this risks making practitioners hesitant to rely on model-agnostic techniques for taking defect-reducing actions [29].

In this paper, in contrast to the current trend of leveraging black-box models, we show that it is possible to provide software improvement plans by mining action rules from historical data and filtering out those that overlap the least with past code changes. Based on this idea, we propose CounterACT, an approach that provides counterfactual explanations based on Action Rule Mining (ARM) [59]. By running simulations over the historical records, CounterACT generates recommendations that help in preemptively reducing the likelihood of potential defects by explaining how an action (a change in the value of one or more flexible attributes) could impact the classification of a given object (e.g., from defective code to working code). Since it is inferred from classification rules mining, the process that generates the plans is fully transparent and a developer can understand why a certain plan was recommended.

To assess the effectiveness of CounterACT, we answer the following research questions:

RQ1: How effective is the rule-based guidance generated by CounterACT? At the release level, the rule-based guidance generated by our approach achieves a high True Positive rate of an average 97% meaning that the learned action rules highly match historical bug fixes done by developers. In addition, the action rules, even when derived from less frequent patterns (as indicated by the average support of 13%), have a high median confidence of 89%. This implies that when a bug fix is recommended by an action rule, there is an 89% chance that the expected non-buggy outcome will occur. Furthermore, the positive uplift values, with an average of 43% indicate that applying the recommended action will increase the likelihood of the desired outcome (reduction of defects) by 43% compared to not taking that action.

RQ2: How does CounterACT compare against competing defect reduction approaches at the release level? To showcase the effectiveness of counterfactual-based planning (using action rule mining) in generating plausible and actionable recommendations to reduce defects in subsequent releases, we compare CounterACT with TimeLIME and other post-hoc defect reduction planners as well as the original ARM algorithm using four performance criteria (i.e., overlap, improvement score, precision, and recall) across 9 projects. Our analysis shows that CounterACT plans have the highest median overlap of 95% which implies that our suggested plans align more closely with actual developer changes when compared to other competing approaches. Additionally, the actions proposed by CounterACT are associated with a larger reduction in defects than other algorithms (median improvement score 91,33%).

RQ3: How does CounterACT compare against competing defect reduction approaches at the commit level? In order to ensure the practicality of our approach, we provide software quality improvement plans at the change level (Just-in-Time or JIT) allowing quicker defect remediation. To determine how well CounterACT performs against other defect reduction planners at a finer granularity, we reuse the performance criteria from RQ2 and evaluate its performance on 5 open-source projects at the commit level. Our findings show that CounterACT also outperforms competing approaches at the commit level, where the overlap score of CounterACT is on average 85.97%. This implies that our plans have the highest similarity with actual actions done by developers in reducing defects at a finer granularity, whereas the performance of the current state-of-the-art approach TimeLIME declines from 79.17% at the release level to 38.36% at the commit level.

Furthermore, the plans provided by state-of-the-art techniques can be challenging to implement by developers because they are reduced in terms of code metrics. In this paper, we discuss the use of Large Language Models (LLMs) coupled with CounterACT plans to generate automated code edits for developers. Our experiments show that the combination of LLMs and planning can lead to enhanced guidance. Indeed, our primary results indicate that the LLM supported by CounterACT plans effectively addresses and resolves bugs in 28 out of 40 cases while the vanilla LLMs recommendations produced incoherent code edits in 26 out of 40 cases.

Overall, our findings suggest that CounterACT is a promising approach for generating plausible and interpretable defect reduction plans, outperforming existing approaches while providing actionable insights for developers. Our contributions are the following:

- We introduce a novel approach CounterACT which leverages action rule mining to yield defect reduction plans.
- We show that CounterACT suggests plans that have higher similarity with developers' actions in reducing defects than TimeLIME's while being fully interpretable.
- We extend the methodology of defect reduction planning by also considering commit-level defect reductions in addition to release-level which was previously studied.
- We use a Large Language Model (CodeLlama [51]) to automate the recommendation of code changes using the plans provided by our approach, as an additional dimension to evaluate the quality of CounterACT's commit-level recommendations.

2 BACKGROUND AND RELATED WORK

2.1 Software Defect Prediction

2.1.1 Machine Learning Techniques. As software projects develop and evolve, defects are introduced, resulting in exponential costs that consume limited software quality budgets and resources. Over the years, numerous approaches and techniques have been proposed to proactively identify and mitigate defects in software systems [25, 33, 38]. Recent advancements have focused on the use of black-box ML models due to their higher predictive accuracy, SQA teams can use SDP models which are trained using historical data to predict the likelihood of defect-prone software modules in the future in order to support decisions about resource allocation in SQA activities [14, 35]. Indeed, SDP models were explored at different levels of granularity such as release level and at change level (just-in-time) [22, 68].

2.1.2 Post-hoc Explanations. As ML models become more complex and achieve better accuracies, understanding their decision-making process becomes much harder and leads to a lack of trust in the predictions. Over the years, many studies used model-agnostic techniques such as Local Interpretability Model-Agnostic Explanations (LIME) [50] and Shapley Additive Explanations (SHAP) [31], to support practitioners in decision-making regarding software defects. Recent studies used the latter in selecting the most impactful features to make the black box ML models more

explainable [13], detecting buggy lines of codes [46], and providing plans to reduce future defect-introducing commits [47].

2.2 Defect Reduction Planning

2.2.1 Planners. Despite efforts to make defect prediction models more explainable, the lack of actionability on the generated explanations is a roadblock toward their wide-spread application [61]. Indeed, even if ML models could provide explanations for their decisions, we would still need to verify that those explanations lead to concrete and applicable recommendations. For instance, if a certain module in a software project is flagged as high risk by a predictive model and a tool, such as LIME [50], identifies its risk factors (e.g. large LOC, large Cyclometric Complexity), a developer might still want to know "what plan can I follow to reduce the risk of defects on that file?".

Henceforth, we will refer to a *planner* p as a function that takes code metrics values m and outputs a region $p(m) = R$ of "safe" code metric values. We will often call the safe region R the *plan* since it highlights where the code metrics should be. Hence the action recommended to the developer by the planner is to change the code so that the new metric values land in the plan.

Before the boom in Machine Learning techniques, defect reduction planners were often provided by empirically chosen thresholds for code complexity metrics [2, 41, 55]. The main intuition behind these approaches is that large values of code complexity metrics are indicative of bad coding practices and poor maintainability, which leads to more defects [66]. Therefore, if a code metric m has a value beyond a threshold γ, then these defect reduction planners recommend refactoring the code so that the metric lies under the threshold.

$$p(m) = \begin{cases} [0, \gamma] & \text{if } m > \gamma \\ m & \text{otherwise.} \end{cases} \tag{1}$$

The different approaches differ in how the thresholds γ are computed. For instance, Alves planner [2] computes it with a weighted Cumulative Distribution Function (CDF) while Shatnawi [55] computes it with a Logistic Regression. Additionally, Oliveira [41] find values of r and γ s.t. $r\%$ of the classes have metric values $m \leq \gamma$. The values of r and γ are obtained via an optimization problem that trades-off idealized and realistic code practices.

These three approaches have shown certain limitations: (1) they do not take into consideration historical code changes, and (2) they do not employ the full arsenal of highly predictive ML techniques available today. Therefore, they can be inaccurate or lead to plans that are surprising to developers and impossible to apply in practice.

To address these two points, two ML-based defect reduction planners have recently been proposed: XTREE [27] and TimeLIME [45]. The former fits a decision tree classifier on software defect data and provides plans that change the prediction of the model while acting on code metrics that tend to historically vary together. The latter employs the post-hoc explainability technique LIME [50], to identify the code metrics that increase the defect risk, and derives plans from these metrics while ensuring a high overlap with past developer changes. TimeLIME [45] has shown to provide better plans than previous approaches in terms of overlap with past changes and correlation with defect reduction. Hence it is the main baseline against which we compare our approach.

Although TimeLIME can provide maintainable defect reduction plans, its main limitation is that the pipeline which generates the plans is opaque. Indeed, using TimeLIME requires first fitting a complex ML model and then approximating it locally with a linear model to get a recommendation. It is therefore difficult to understand why a certain plan was suggested and not another. Furthermore, if all provided plans are bad, the source of the error can be hard to pinpoint since it can be attributed to both the black box model and the explainer. In this work, we will show *transparent* defect reduction planners are possible, thus removing these sources of confusion.

Table 1. Example of computing overlap using the Jaccard similarity function. Plans that match the developer actions are marked gray

	AMC	LOC	LCOM	CBO
Current release t	2.13	504	0.9	5
R for release $t+1$	no change	(470,520)	(0, 0.4)	no change
Next release $t+1$	2.13	530	0.3	2
is a Match?	y	n	y	n

2.2.2 Plan Quality Assessement. To compare different planners we need to define what is a *good* plan. Ideally, the best plan would be the one that leads to the most defect reductions when the developers follow it. However, like previous work [27, 45], we are performing an *observational* study by investigating historical codes changes. We do not explicitly tell the developers to follow our plans. Thus, we cannot claim that any of the plans can *cause* a reduction in software defects, or that they can *fix* bugs. Rather, the main hypothesis behind plans is that code metrics are proxies for good software design practices which influence the prevalence of future defects. Hence, by changing code metrics values following a plan, we only expect developers to improve the design of their code which should prevent *future* defects. Thus,

A good plan should overlap with past code changes and be statistically associated with defect reductions.

This is the main philosophy behind the K-test [26]. More specifically, the K-test examines code regions common to three consecutive software releases $t-1$, t, and $t+1$. This region, indexed by i, could be an object-oriented class, a function, or a file present across all releases. For each version of the software, we must also have access to M code metrics so that $m_j^{(i)[t]}$ is the jth code metric evaluated on the ith code region on version t. The K-test also assumes the existence of a quality measure of the code region evaluated at each version $Q^{(i)[t]}$. Importantly, if p is a good planner, then quality improvement $Q^{[t]} - Q^{[t-1]}$ should be positively correlated with the **overlap** between the plan $p(m^{[t-1]})$ and the actual code change $m^{[t]}$. The overlap is computed via the formula

$$O(p, m^{[t-1]}, m^{[t]}) = \#\{\, j : m_j^{[t]} \in p\big(\, m_j^{[t-1]} \,\big) \,\}/M, \tag{2}$$

which is the ratio of code metrics from version t that land inside the plan proposed in version $t-1$. In Table 1 we show an example of an overlap score.

Following previous studies [7, 36], we use the Number of Defect in Previous Version t **NDPV** as the quality metric $Q^{[t]}$ and the correlation between quality improvement and overlap denoted as the *improvement score* can be computed:

$$S_{\text{scaled}} = \frac{\sum_{i=1}^{N} \left(Q^{(i)[t]} - Q^{(i)[t-1]}\right) \times O(p, m^{(i)[t-1]}, m^{(i)[t]})}{\sum_{i=1}^{N} Q^{(i)[t]} - Q^{(i)[t-1]}} \tag{3}$$

where the index i again represents a code region (e.g. class) and so we sum over all N regions. This correlation is highly positive if high overlap tends to occur when the quality increases.

To identify if the developers can implement the recommended plans by `CounterACT`, as suggested in prior work [45] we label the generated plans into one of the following categories:

- **True Positive (TP):** The plan recommends changes, which are observed later on.
- **True Negative (TN):** The plan recommends no changes, and no changes are observed later.
- **False Positive (FP):** The plan recommends changes, but they are not observed later.
- **False Negative (FN):** The plan recommends no changes, yet some changes are observed later.

Then we calculate the precision, recall and the F1-score, as follow:

- **Precision = TP/TP+FP**: Among all the recommended changes by the defect reduction planner, how many are observed in the next release?
- **Recall = TP/TP+FN**: Among the changes observed in the next release, how many of them match the defect reduction planner's recommendations?
- **F1-score = 2x(Precision*Recall)/(Precision+Recall)**

Indeed, a higher precision indicates that the suggested plans are more frequently utilized, while a higher recall implies fewer unexpected changes in the next release. Consequently, effective defect-reducing plans should be associated with both high precision and high recall.

3 ACTION RULE MINING

An action rule describes how an action (a change in the value of one or more flexible attributes) could impact the classification of a given object. It serves as a recommendation for a course of action to take, to increase the probability of a desired class (a counterfactual "what if" explanation for the classification). Action Rules mining [59] has two phases: (1) Classification rules mining; (2) Generation of Action Rules from a subset of discovered classification rules.

Classification rules are mined using Apriori modified for classification rule mining [19]. They take the following form

$$r_c = [(a_1 \wedge b_1 \wedge c_1 \wedge e_1) \Rightarrow d_1], \tag{4}$$

where the *antecedent* a_1, b_1, c_1 and e_1 represent feature values and the *consequent* d_1 denotes the value of the target variable predicted. Each identified rule comes with support and confidence values, indicating its quality. First, support sup(antecedent $\Rightarrow$ consequent) represents the ratio of data instances matching both the rule's antecedent and consequent. Secondly, confidence conf(antecedent $\Rightarrow$ consequent) $= \frac{\text{sup(antecedent}\Rightarrow\text{consequent)}}{\text{sup(antecedent)}}$ is the ratio of instances fulfilling both antecedent and consequent to those satisfying only the antecedent. See Figure 1 for an example. We have two classification rules that predict different outcomes: bug and no-bug. Looking at the rule $[(\text{avg_cc} > 1.4) \wedge (\text{cbo} > 14)] \Rightarrow \text{bug}$, the support is 6% meaning that 6% of the data points have the specified metric values and are also buggy. The confidence is very high (91%) meaning that, out of all the instances that respect the antecedent, a majority of them are indeed buggy.

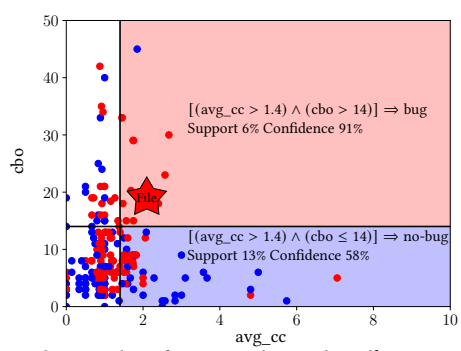

Fig. 1. How action rules are mined. First classification rules with sufficient support and confidence are mined using Apriori. This leads to the two rules in the red and blue regions. Then, rules with matching antecedents but different consequents are combined leading to the action rule with support(6%) and confidence(52%) $r = [(\text{avg_cc} > 1.4) \wedge (\text{cbo} > 14 \rightarrow \text{cbo} \leq 14)] \Rightarrow [\text{bug} \rightarrow \text{no-bug}]$.

Rules are interpretable by design but they describe *what is*, and not *what ought to be*. This is where action rules come into play. For example, To build an action rule, two rules with matching

antecedents but different consequents are combined:

$$\begin{aligned} \text{(First classification rule)} \quad r_1 &= [\omega \wedge \alpha] \Rightarrow \text{bug} \\ \text{(Second classification Rule)} \quad r_2 &= [\omega \wedge \beta] \Rightarrow \text{no-bug} \\ \text{(Action Rule)} \quad r &= [\omega \wedge (\alpha \rightarrow \beta)] \Rightarrow [\text{bug} \rightarrow \text{no-bug}]. \end{aligned} \tag{5}$$

In this equation, ω represents a constant condition (stable attribute), with $(\alpha \rightarrow \beta)$ being the suggested change to a subset of flexible attributes. (bug $\rightarrow$ no-bug) symbolizes the desired change in terms of prediction. Going back to the example of Figure 1, the two rules presented can be combined into the action rule $a = [(\text{avg_cc} > 1.4) \wedge (\text{cbo} > 14 \rightarrow \text{cbo} \leq 14)] \Rightarrow [\text{bug} \rightarrow \text{no-bug}]$ which suggests diminishing the Coupling Between Objects (CBO) of the current class. The metrics used to assess the quality of an action rule differ from those of a classification rules. Indeed, the support is defined as the minimum support between the two rules combined:

$$\text{supp}(r) = \min\{\text{supp}(r_1), \text{supp}(r_2)\}, \tag{6}$$

while the confidence becomes the product of the two confidences:

$$\text{conf}(r) = \text{conf}(r_1) \times \text{conf}(r_2). \tag{7}$$

According to literature [10], the intuition behind this new definition of confidence is as follows: consider all pairings between examples that respect the antecedent of r_1 and those that respect the antecedent of r_2. Then, compute the number of pairs where the label switches from bug to no-bug. This ratio is the confidence of the action rule. Lastly, the uplift [28] of an action rule is defined as the measure predicting the incremental response to an action. It is the difference in decision probability with and without treatment. Instances are categorized into control and exposed groups, with the former exposed to the recommended action and the latter not.

4 METHODOLOGY

Overview. We introduce a novel defect reduction planning approach named CounterACT, based on Action Rule Mining (ARM) leveraging Counterfactuals. The ARM algorithm [59] is designed to mine rules on nominal data, hence our approach adds data discretization using entropy [11] to transform numerical values into tuples which enables the processing of numerical data, expanding the applicability of the approach. In addition, we add actionable analysis detailed in 4.1 for the selection of actionable features. ARM generates multiple defect reduction plans when given a defective instance. Thus, we augment the algorithm with a plan selection algorithm presented in Section 4.3 that leverages historical data based on overlap to ensure precedence and improve the plausibility of plans. We benchmarked the performance of CounterACT on the release level against

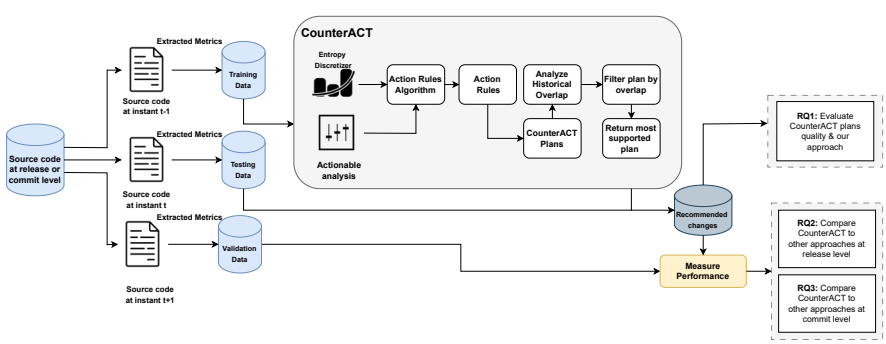

Fig. 2. CounterACT: Overview of the approach. Note that for evaluating other benchmark planners, the grey area be replaced by the corresponding planner.

the original ARM algorithm and several state-of-the-art approaches: model-agnostic techniques such as TimeLime [45] and Lime [17], XTree [27], and statistical methods including Alves, Shatnawi, and Oliveira [2, 41, 55]. In addition, we ventured a step further, examining CounterACT's planning at the commit level and measuring its performance with regard to the aforementioned approaches.

Figure 2 and 3 illustrate an overview of the CounterACT approach. It consists of the following steps: (1) Actionable analysis; (2) Mining action rules from historical data; (3) Plan selection.

4.1 Actionable Analysis

Building upon the foundations set by previous research [16, 45], which mainly restricts the generated recommendations to the attributes that were seen to be frequently modified within the history of a software project. We operate under the assumption that any suggested alterations to a code metric should align with observations from historical records.

Consequently, we generate the actionable set encompassing the top-M features with the highest variance in historical data. Here, the user has the flexibility to specify the value of M. In the context of our experiments, the selection of the top-M features is guided by evaluating the magnitude of changes observed between two consecutive releases $(t-1, t)$. This is determined by measuring the hedge g value [24], which is calculated as:

$$g = \left(\mu^{[t-1]} - \mu^{[t]}\right)/S_{\text{pooled}}, \tag{8}$$

and

$$S_{\text{pooled}} = \sqrt{\frac{(n^{[t-1]}-1)(\sigma^{[t-1]})^2 + (n^{[t]}-1)(\sigma^{[t]})^2}{n^{[t-1]}+n^{[t]}-2}}, \tag{9}$$

Here, the variables $\mu^{[t-1]}, \mu^{[t]}, \sigma^{[t-1]}, \sigma^{[t]}$ represent respectively the means and standard deviations of an attribute in two consecutive releases. The terms $n^{[t-1]}$ and $n^{[t]}$ denote the sample size for each release. Moreover, since a plan is generated using the set of top-M features with the most changes, we denote the latter as the actionable set. This means that the action rule algorithm will mine rules altering only the features in that set.

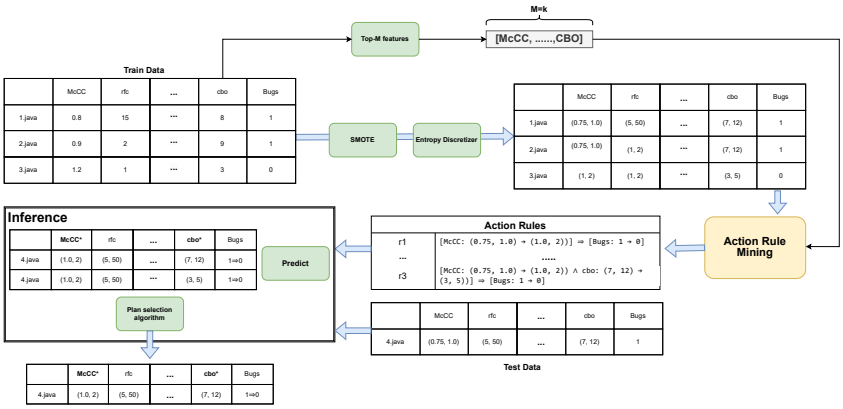

Fig. 3. CounterACT: an illustrative example

4.2 Mining Action Rules

Motivation. Among the variety of recommendation techniques developed by the ML community, Action Rules [59] are most relevant to our problem for multiple reasons. First, most Counterfactual methods return a single perturbed instance x' [42, 65] or a set of k perturbed instances $\{x'_i\}_{i=1}^{k}$ [37, 53].

However, in the context of software engineering, we want recommendations from x to x' to match past changes in code. If the code metrics stored in the vector x are numerical, then it is not realistic to expect to perceive a change that goes exactly from x to x' in the code history. To ensure that overlaps with previous code changes are non-zero, like previous defect reduction planners proposed in software engineering, it is better to provide a recommended *range* of feature values. For example, instead of recommending changing a code metric x from $x = 34$ to $x = 25$, we want to recommend reducing $x = 34$ to a range $10 \leq x \leq 25$, which is exactly what action rules provide. Secondly, the plans generated by action rule miners are transparent. Indeed, since these action rules are derived from classification rule mining, their computation is well documented [19], and, assuming they are small enough, the rules could be visualized and understood by software developers.

Since the action rule mining algorithm takes discretized ranges of attribute values, drawing inspiration from previous studies such as TimeLIME [45] and LIME [50], we use an entropy-based discretizer [11] which provides a discretized interval indicating the range of values during which the feature will maintain the same effect to the prediction result.

After obtaining the set of actionable features, we set them as flexible attributes in the algorithm of action rule mining. Then, we train the algorithm on historical releases (RQ2) or commits (RQ3). We use the original algorithm's default values for the minimum support (5%) and minimum confidence (55%). The latter refers to those of the classification rules from which action rules are generated.

Quality assessment of action rules. In addition to the evaluation of the generated action rules in terms of support, confidence, and uplift, we further extend the assessment by calculating the following metrics:

- **False Positive Action Rule (FP):** An action rule is considered **FP** if the suggested action for the metrics has been historically observed not to fix a bug. In other words, if the historical change between versions $t-1$ and t does not resolve a bug in a file and the action rule matches this change, then the rule is considered as a **FP**.
- **True Positive Action Rule (TP):** if the suggested action for the metrics has been historically observed to fix a bug. In other words, if the historical change between versions $t-1$ and t did resolve a bug in a file and the action rule matches this change, then the rule is considered as a **TP**.

4.3 Plan Selection

The last step of the action rule mining algorithm is inference. In which we input a defective instance and the algorithm returns a recommended action on each software attribute inferred from the previously mined action rules. We present two examples of defect reduction plans (p_1), (p_2):

$$\begin{aligned}
&\text{(Action Rule)} \quad p1 \;= [NOC_class(0, 0.5) \wedge NUMPAR_method((3.0, 4.0) \rightarrow (1.0, 3.0)),\\
&\qquad NL_method((0.5, 1.0) \rightarrow (0.0, 0.5)),\\
&\qquad McCC_method((0.99, 1.0) \rightarrow (0.0, 0.99))] \Rightarrow [True \rightarrow False] \qquad (10)\\
&\text{(Action Rule)} \quad p2 \;= [NOC_class(0.5, 1.0) \wedge NOI_method((0.0, 0.5) \rightarrow (0.0, 1.0)),\\
&\qquad NUMPAR_method((3.0, 4.0) \rightarrow (1.0, 3.0)) \Rightarrow [True \rightarrow False]
\end{aligned}$$

Since we have multiple plans, we establish a plan selection process that leverages the similarity to historical records by aligning the selected plans more closely with the proven historical patterns of defect bug reductions. Initially, a set of candidate plans is generated. Then we measure the overlap of each plan with the historical logs using Equation 2 and return the plan with the highest overlap score.

Algorithm 1: Plan Selection Algorithm

```
Input: Candidate plans generated_plans and non-buggy historical data past_fixes
Output: The proposed plan for the instance
plan ← None;
pool_overlap_score_list ← list();
for plan_candidate in generated_plans do
    overlap_score_list ← list();
    for fix in past_fixes do
        overlap_score ← Overlap(plan_candidate, fix);
        append overlap_score into overlap_score_list;
    end
    plan_overlap ← median(overlap_score_list);
    append plan_overlap into pool_overlap_score_list;
end
max_overlap_index ← random index of max value in pool_overlap_score_list;
plan ← element at max_overlap_index from generated_plans;
```

Moreover, we perform an analysis to assess the impact of different plan selection strategies for the buggy code instances. Our approach generates multiple defect reduction plans, thus we evaluate the outcomes of selecting a plan: (a) at random, (b) based on the highest action rule support, (c) based on the highest action rule confidence, and (d) our proposed plan selection. In Figure 4, we evaluate the different plan selection approaches and report the median overlap score. On average, our plan-selection approach results in the highest median overlap score of 86.14%. In comparison, relying on the action rule support results in a median overlap score of 85.5%.

Figure 4 shows that our plan selection algorithm performs better overall. In addition, after conducting Wilcoxon signed rank statistical test [67] to compare the values of the various strategies against the 'CounterACT plan selection', the results indicate that 'Random' and the 'action rule confidence' values both show statistically significant differences from the 'CounterACT plan selection', with p-values less than 0.05. However, the 'action rule support' values do not show a statistically significant difference from 'CounterACT plan selection', as evidenced by a p-value of approximately 0.3123. While the overlap of CounterACT's plan selection is not significantly higher than the 'action rule support' we still favor it, because it does generally provide better overlap score and more stable results as evidenced by the lower standard deviation. For a comprehensive breakdown of the underlying algorithm to select the most supported plan, please refer to Algorithm 1.

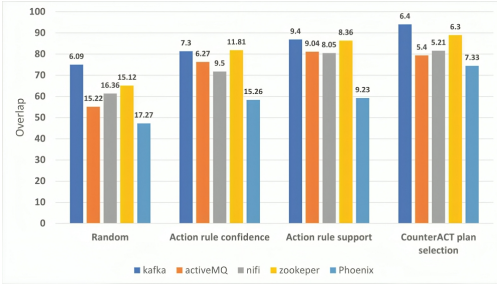

Fig. 4. Median overlap scores between different plan selection methods. The reported values on the histograms are the standard deviations

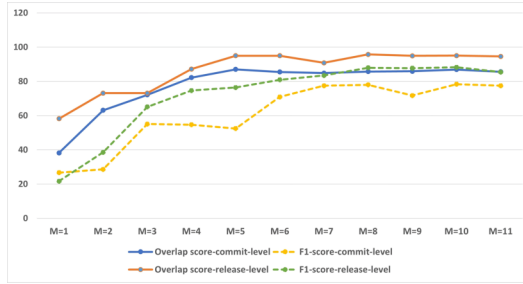

Fig. 5. Results of top-M experiments. We chose the value of $M = 10$, since it shows the most stable overlap and F1 score

5 EXPERIMENTAL DETAILS

The experiments report the performance of CounterACT and other state-of-the-art approaches by comparing the quality of plans recommended by each approach at the release and commit levels.

5.1 Release Level

Studied projects. To conduct our study, we use a publicly available defect dataset used in previous work [27, 45] to support verifiability and foster replicability. The data consists of 9 Apache open-source projects collected by Jureczko et al [21] as described in Table 2 (Camel was used twice since we have 4 releases). It is necessary for the data to contain a minimum of three consecutive releases for each project since the main hypothesis of our approach is that the guidance which is derived from past knowledge (a release t-1) can be used to provide plans for defective files in the target releases (a release t) and be applicable to prevent software defects in future releases (a release $t+1$).

Table 2. The dataset of defects used. The last column displays the decrease in bugs found in identical files when comparing the test release to the validation release. A negative number in this row suggests that the subsequent validation release had a higher bug count than its predecessor.

Project	Training (oldest)	Testing (newest)	Validation (most recent)	#files	# matched files	# bugs in testing set	# bug in validation set	# bugs reduced
Jedit	4.0	4.1	4.2	367	78	216	74	142
Camel	1.0	1.2	1.4	872	210	508	247	261
log4j	1.0	1.1	1.2	205	35	83	120	-37
Xalan	2.5	2.6	2.7	885	385	529	381	148
Ant	1.5	1.6	1.7	745	91	183	163	20
Velocity	1.4	1.5	1.6	229	138	321	144	177
Poi	1.5	2.5	3.0	442	247	495	366	129
Synapse	1.0	1.1	1.2	256	58	97	65	32

Table 3. Software metrics.

Release level		Commit level		
Metric	**Description**	**Granularity**	**Metric**	**Description**
WMC	Number of methods	**Class**	CBO	Coupling Between Object classes
DIT	Depth of Inheritance Tree		TNG	Total Number of Getters
NOC	Number of Children		NLPA	Number of Local Public Attributes
CBO	Number of objects the class is coupled with		NLE	Nesting Level else if
RFC	Response for a Class		NOD	Number of Descendants
LCOM	Lack of cohesion in methods		NII	Number of Incoming Invocations
CA	Number of Afferent Couplings		TNLS	Total Number of Local Setters
CE	Number of Efferent Couplings		NOP	Number of Parents
LCOM3	Normalized measure of Lack of Cohesion		LCOM5	Lack of Cohesion in Methods 5
LOC	Size of the Java Byte Code		CI	Clone Instances
DAM	Data access		NUMPAR	Number of Parameters
MOA	Number of attributes which are user-defined classes	**Method**	CI	Clone Instances
MFA	Ratio of method which are inherited		McCC	McCabe's Cyclomatic Complexity
NPM	Number of Public Method (API Size)		NOI	Number of Outgoing Invocations
CAM	Cohesion Amongst Classes		NLE	Nesting Level else if
IC	Inheritance coupling		NII	Number of Incoming Invocations
CBM	Coupling between methods		LLOC	Logical Lines of code
AMC	Average method complexity		TNS	Total Number of Statements
max_CC	Maximum McCabe's cyclomatic complexity seen in class			
avg_CC	Average McCabe's cyclomatic complexity seen in class			

Software metrics. The data consists of records of previous defects for software modules (e.g. classes) for which there are 20 static code metrics that quantify the aspects of software design as shown in Table 3. They are a combination of McCabe [32] and complexity metrics in conjunction with extended CKOO metrics [6] and Lines of Code (LOC). Previous studies indicate that these measures correlate highly with the likelihood of defects and provide a comprehensive analysis of the code complexity and object-oriented aspects [20] where they analyzed the significance and correlation of each software metric with defect numbers.

Actionable analysis. In order to get the set of actionable attributes, we input historical data from the older releases to compute the variance of each feature. Then we select the top-M features with the largest variance as the actionable set of flexible attributes. Recommendations based on other features are ignored. The parameter M can be user-specified and the features may vary with respect to different projects and the releases used as historical data. Here we set the default value of M to be 10, which means 50% of all twenty features can be mutated.

We experiment with various values of M to determine the one that yields the best overlap score as an evaluation metric on the release and commit level data, ensuring an optimal setting is found through exhaustive evaluation. As shown in Figure 5, our results suggest that M = 10 is a useful default setting as it provides the more stable median F1-score (88%) and highest overlap (95%) for the release level data, and yields a stable median F1-score and overlap score with respective values of 78% and 85.97% for the commit level data.

Mining action rules. First, we use an over-sampling technique named SMOTE (Synthetic Minority Oversampling Technique) [5] since we work on imbalanced datasets where defective instances constitute only a minor proportion of the entire data set. As suggested by prior work [1, 60] who found that the SMOTE technique outperforms other class rebalancing techniques, SMOTE finds its nearest neighbors in the feature space for each instance in the minority class. Then, it randomly selects some of these nearest neighbors and generates synthetic instances. It does this by interpolating between a minority class instance and its selected neighbor. Finally, SMOTE combines the synthetic instances with the undersampling of the majority class to produce a set of balanced instances. We use the implementation of SMOTE as provided by Imbalanced-Learn Python library [30]. This step is crucial because previous research on the studied dataset has highlighted the difficulty of building predictors when dealing with small target classes [1].

Next, we use an entropy-based discretizer on the data and train the action rule mining algorithm on release t-1 and release t. Then we conduct inference, where we generate defect reduction plans on the defective instances of release t and then compute the overlap with actual changes made by developers in the future release t+1.

After getting defect reduction plans from the planners, we assess their performance using the overlap score as described in Section 2. In addition, to assess whether a plan has a high/low similarity to a bug reducing/adding actions. We measure the improvement score, also detailed in Section 2. Given that the total number of bugs varies from each project as shown in Table 2, we compare the results proportionally to their respective project. Indeed, we expect a project with more bugs reduced in the validation dataset (i.e release t+1) to result in a planner that scores higher than a dataset with fewer bugs reduced.

5.2 Commit Level

In this section, we detail the experimental setup for examining CounterACT's planning at the commit level, leveraging counterfactual-based planning at a finer-grained level (change-level).

Data. We select just-in-time defect datasets from the 2021 PROMISE file-level just-in-time defect prediction challenge [64]. Thus, our data contains one instance for every file that was changed within each commit. We choose to work with this data since it consists of different large-scale open-source Apache projects and the labels were assigned based on specific criteria: commits were tagged as bug-fixing if they included a link to a Jira issue, the issue was identified as a bug, and the fastText classification method [15], with a 95% recall rate, verified this. The authors of the data used git blames to determine the inducing changes for each changed line in a Java file using the SZZ algorithm [58]. To avoid selection bias, We randomly select 5 projects, as detailed in Table 4.

Commit features. The dataset span across 5 categories of many fine-grained features: Just-In-Time (JIT) [23], Fine-Grained JIT [43], PMD features, Warning density features, and static code

Table 4. A summary of the studied commit defective datasets provided by PROMISE challenge.

Projects	Training Data				Testing Data			
	Start Date	End Date	# Commits	# Defective Commits	Start Date	End Date	# Commits	# Defective Commits
nifi	08/12/2014	17/02/2017	16445	1557 (9.4%)	17/05/2017	29/09/2017	1093	145 (13.2%)
zookeper	03/11/2007	01/07/2013	3545	422 (11.9%)	07/10/2013	11/09/2017	850	89 (10.4%)
kafka	01/08/2011	15/03/2017	6742	818 (12.1%)	16/06/2017	29/09/2017	1244	99 (7.9%)
Phoenix	27/012014	17/08/2016	8833	1486 (16.8%)	17/11/2016	28/09/2017	1094	196 (17.9%)
ActiveMQ	12/12/2005	11/04/2016	28570	3568 (12.4%)	11/07/2016	21/09/2017	1051	73 (6.9%)

metrics. We focus on static code metrics since the criteria we use are that (1): static code metrics are widely used in defect prediction [63], (2): in our study, we do not only look to provide to explain "why" an instance was defective but we also provide "what to do". For example, while JIT features are widely used for commit-level defect prediction studies, static code metrics provide a descriptive analysis of the code and how complex its structure gets. We adopt code metrics that cover the source code properties of complexity, coupling, documentation, inheritance, and size as shown in Table 3. Moreover, we mitigate collinearity and multi-collinearity by using AutoSpearman [18], an automated feature selection approach. After using the latter, we finally select 18 features that are not highly correlated with each other. To handle the class imbalance we apply SMOTE, only on the training dataset.

Actionable analysis. Following the same approach as our release level analysis, we set the number of flexible attributes to the action rule mining algorithm using the top M=10 features with the highest variance between windows of commits. We first sort the commits by date. Subsequently, we partition the data into three equal chunks. We then compute the variance for each feature between these chunks. Note that we set the number of changed features of the other planners the same as CounterACT planner to ensure fair comparison for both release and commit level experiments.

Mining action rules. In our study, we proceed as follows to split our data into train, test, and validation sets:

- Training Set: we start by chronologically ordering all commits. We excluded commits from the most recent three-month window to avoid the inclusion of recent buggy commits that may lack associated fixes. Following this, we extracted the last 250 commits (in alignment with the criteria proposed by the PROMISE challenge) to create our test set. All the historical commits represent our training set.
- Test Set: This split is composed exclusively of commits identified as "buggy" from the reserved 250 commits.
- Validation Set: For each "buggy" commit within the test set, we identify a corresponding "fix" commit as follows: The "fix" commit should relate to the same file as its associated buggy commit. The "fix" commit must have a commit date succeeding that of its corresponding buggy commit. Finally, the identified ‘"fix" commit must be marked as non-buggy.

6 RESULTS

(RQ1): How effective is the rule based guidance generated by CounterACT? In figure 6 we illustrate the distribution of support, confidence and uplift of the action rules generated on the release level of all projects. CounterACT produces rules characterized by high confidence (average 89%), indicating a high conditional probability that when a bug-reducing action is recommended by the action rule, the predicted non-buggy outcome is likely to occur in 89% of cases. However, average support shows that approximately 13% of the observed bugs have a specific associated recommended fix. Given that the support values range from 10.12% to 29.18%, this suggests that some fixes are applicable to a wide range of bugs, while others might be more specific or less common. From this, we can infer that many rules, while specific and relevant to a minor group of bugs, remain trustworthy because of the high confidence under specific conditions. Moreover,

the rules tend to have a high uplift value of 43% and we observe that all the values are positive. Following our definition in section 3, the uplift indicates that the occurrence of the action to fix a bug (treatment) increases the chances of the bug fix happening more than if we did not apply any action. This means that the rules are not only capturing frequent bug fix actions but also actions that have a meaningful relationship with the outcome.

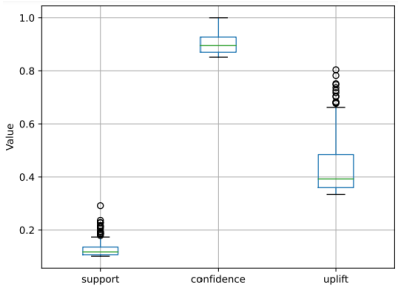

Fig. 6. Support, Confidence, Uplift distribution at the release level

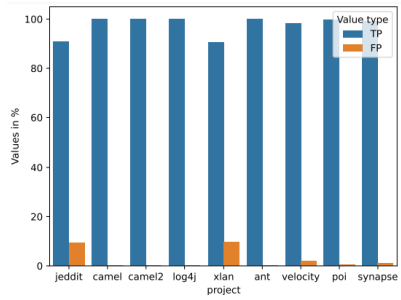

Fig. 7. The True Positive (TP) and False Positive (FP) of the learned rules

The majority of learned action rules are associated with bug fixes. In Figure 7, we display the True Positive (TP) and False Positive (FP) rates for the learned rules across all projects. We observe that CounterACT achieves a TP rate higher than 80% for each project. Indeed, this implies that the action rules mined by CounterACT are often present in the history of actual bug reduction plans used by developers.

> CounterACT produce rules that, while based on less frequent patterns (13%), are highly predictive (high confidence of 89%) and indicate a meaningful relationship between antecedent and consequent (significant uplift of 43%). Additionally, the learned action rules are highly associated with historical bug fixes done by developers (average TP 97.56%).

(RQ2): How does CounterACT compare against competing defect reduction approaches at the release level? The recommendations provided by CounterACT align more closely with developers' actions compared to competing approaches. This is evidenced by a higher overlap score : 95% on average. Tables 5 and 7 demonstrate how frequently code changes intersect with CounterACT's suggested plans within projects. The median overlap score, $O(p, m^{[t]}, m^{[t+1]})$, for each planner across all instances are reported in Table 5. In Table 7, we present the interquartile range (IQR) for overlap scores among all plans. When median overlap scores are similar, a lower IQR signifies greater stability and robustness in the defect reduction planner. CounterACT shows a small IQR (5 on average) while maintaining the highest median overlap score in 8 out of 9 projects when compared to TimeLIME, the current state-of-the-art. This shows that using action rules generates robust plans and prevails over other defect reduction planners in terms of providing plans that better resemble developers' choices, all without relying on a black box model. Thus, CounterACT is designed to provide achievable and maintainable solutions, which is not the case for ARM algorithm. As shown in Table 5, the ARM algorithm on its own was unable to generate rules for 5 out of 9 projects on the release level, having an median overlap score of 31.55%.

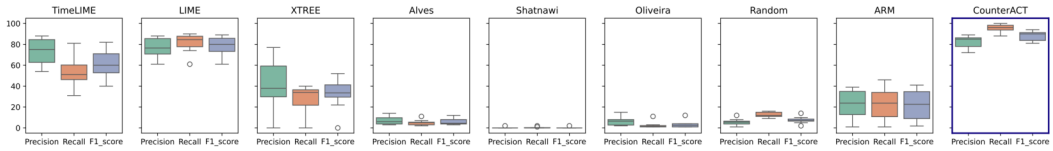

Fig. 8. Precision, Recall and F1-score distribution for each planner at the release level

Table 5. Median overlap scores. A higher scores being preferable and indicated by dark shade

	TimeLIME	LIME	Shatnawi	Alves	Random	Oliveira	XTREE	ARM*	CounterACT
jedit	70	60	50	30	17	35	75	-	95
camel1	90	85	65	60	15	62	80	34.27	95
camel2	80	75	50	50	25	55	75	32.67	100
log4j	75	75	55	45	25	50	65	-	95
xalan	100	100	75	80	65	85	75	38.09	100
ant	65	55	35	35	25	35	65	-	90
velocity	90	85	70	55	20	75	80	21.18	95
poi	85	70	55	40	0	40	75	-	95
synapse	67	70	40	40	10	42	62	-	90
AVG	79.17	75.56	55	48.33	22.55	53.33	72.22	31.55	95.00
STD	11.86	14.24	13.23	15.21	17.94	17.81	7.12	7.27	3.54
p-value**	0.01	0.01	0.003	0.003	0.003	0.003	0.003	0.001	-

*Action Rule Mining
**Wilcoxon signed rank test to compare with CounterACT

Table 6. Average S_{scaled} score is presented. The top-performing planner is indicated by dark shade.

	TimeLIME	LIME	Shatnawi	Alves	Random	Oliveira	XTREE	ARM*	CounterACT
jedit	69	59	50	31	19	34	71	-	91
camel1	79	75	57	51	15	55	75	33	92
camel2	68	56	37	31	13	42	71	29	81
log4j	74	71	48	43	23	44	61	-	87
xalan	92	88	65	73	59	81	75	35	98
ant	82	72	59	46	40	60	69	-	100
velocity	84	78	58	49	19	58	74	19	94
poi	79	73	61	50	0	51	74	-	93
synapse	63	68	35	35	8	41	52	-	86
AVG	76.67	71.11	52.22	45.44	21.78	51.78	69.11	29	91.33
STD	9.06	9.60	10.57	13.02	17.75	13.96	7.77	7.11	5.96
p-value**	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.001	-

*Action Rule Mining
**Wilcoxon signed rank test to compare with CounterACT

The majority of recommendations proposed by CounterACT are implemented in the next release (high precision), and they are less likely to adopt changes that haven't been suggested (high recall). In Figure 8, we display the distribution of precision, recall, and F1-scores of plans defined in 2. The results reveal that CounterACT defect reduction planner achieves the highest precision score with 82.3%, indicating that developers frequently implement CounterACT's plans. Moreover, Figure 8 presents the distribution of recall of plans, where a low value implies the presence of more unanticipated changes. CounterACT exhibits the highest recall (95.5% on average) among all defect reduction planners, suggesting that it provides a variety of plans that are reflected in developers' actions. In fact, our data shows that developers are unlikely to implement changes that have not been recommended.

CounterACTs' plans exhibit the highest correlation between overlap and bug reductions in all projects. Table 6 displays the average S_{scaled} score (improvement score), where we can see that CounterACT attains the highest score across all projects : average S_{scaled} of 91.33%. Thus, when the overlap between the plan and actual changes is high, there tends to be an increase in code quality. Conversely, when the overlap is low, there tends to not be an increase in code quality.

Crucially, this does not prove that CounterACT *fixed* more bugs than other planners. Making such a claim would require an *interventionnal* study where we tell developers to follow the various plans. Rather, a higher correlation means that the plans are less surprising since they overlap more with historical code changes that decrease the number of defects.

Furthermore, we conduct a statistical analysis of the empirical results, using the non-parametric paired Wilcoxon signed rank tests [67] since the samples compared among planners are dependent. The null hypothesis of such tests is that CounterACT is as likely to yield a high-quality plan as any other planner. As shown in Table 5, and Table 6, the p-value is less than 0.05 in all cases. This suggests that there is a statistically significant difference between the performances of CounterACT and the other approaches. Hence, there is strong evidence to reject the null hypothesis. Moreover, using the Wilcoxon signed rank tests across the different F1-scores, CounterACT demonstrates statistical significance compared to the other defect reduction planners with p-values consistently below the 0.05 threshold.

At the release level, CounterACT plans are associated with greater defect reduction compared to competing approaches with an average S_{scaled} score of 91.33%. On average CounterACT plans overlap with actual developers' changes more than other approaches (95%). Finally, CounterACT achieves the highest recall (95.5%) and the best average F1-score (88.12%), showing that developers would likely be able to implement its plans.

Table 7. Interquartile range (IQR) overlap scores at release level: lower IQRs are preferred and are indicated by dark shade.

	TimeLIME	LIME	Shatnawi	Alves	Random	Oliveira	XTREE	ARM*	CounterACT
jedit	15.0	25.0	20.0	20.0	15.0	20.0	10.0	-	10.0
camel1	25.0	35.0	40.0	50.0	25.0	50.0	10.0	14.0	5.0
camel2	20.0	35.0	30.0	45.0	30.0	50.0	6.25	15.0	5.0
log4j	27.5	12.5	35.0	37.5	30.0	40.0	20.0	-	10.0
xalan	15.0	20.0	15.0	30.0	35.0	45.0	0.0	5.0	5.0
ant	25.0	27.5	20.0	15.0	17.5	20.0	20.0	-	10.0
velocity	10.0	30.0	25.0	35.0	15.0	50.0	5.0	15.0	0.0
poi	15.0	15.0	15.0	20.0	10.0	20.0	10.0	-	0.00
synapse	38.75	25.0	30.0	38.75	15.0	35.0	23.75	-	11.25

*Action Rule Mining

Table 8. Median overlap scores (percentages) at the commit level. Higher scores are preferable and indicated by darker shading

	TimeLIME	LIME	Xtree	Alves	Shatnawi	Oliveira	ARM*	CounterACT
kafka	36.36	45.45	35.35	36.36	9.09	54.55	21.18	94.0
activeMQ	18.18	36.36	27.27	27.27	27.27	45.45	34.78	79.34
nifi	45.45	31.82	36.36	13.64	9.09	18.18	26.31	81.58
zookeper	45.45	27.27	24.24	13.64	9.09	18.18	28.09	88.95
phoenix	27.27	36.36	18.18	27.27	0.00	36.36	31.81	74.47
AVG	38.36	35.25	28.08	23.46	10.09	34.55	28.43	85.97
STD	13.68	6.09	7.33	9.04	9.598	15.26	5.21	6.4
p-value**	0.03	0.03	0.03	0.03	0.03	0.03	0.03	-

*Action Rule Mining
**Wilcoxon signed rank test to compare with CounterACT

(RQ3): How does CounterACT compare against competing defect reduction approaches at the commit level? CounterACT is able to capture the relationship between bugs and software code metrics, even at a finer granularity (i.e., at the commit level). In Table 8, we display how often the plans recommended by CounterACT overlap with code changes. CounterACT plans present the highest median overlap score (85.97% on average), nearly twice that of the next highest competing approach TimeLIME (38.36% on average) while the original ARM algorithm yields a median overlap score of 28.43%.

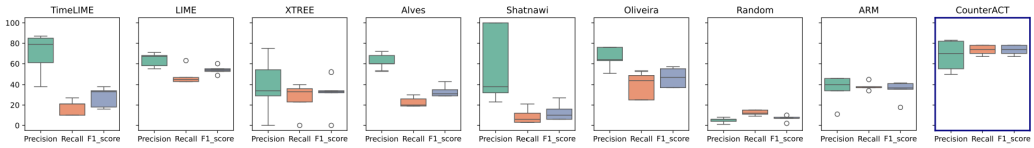

Fig. 9. Precision, Recall and F1-score distribution for each planner at the commit level

The majority of recommendations proposed by CounterACT are adopted in future commits (high precision), and they are less likely to adopt changes that haven't been suggested (high recall) at a finer granularity. In Figure 9, we display the precision and recall of plans for various defect reduction planners. The results reveal that CounterACT achieves not only the highest scores in both precision and recall with respective median values of 78% and 75% but also the most optimal precision-recall trade-off (average F1-score 72.83%), indicating that developers can potentially implement the majority of CounterACT's recommendations. In fact, developers are less likely to implement changes that have not been recommended.

In summary, CounterACT produces stable plans by keeping a consistent performance at both release and commit granularities (median overlap score of 95% and 85.97% respectively), outperforming other defect reduction planners in terms of providing plans that align more closely with developers' choices. This suggests that counterfactual analysis is an efficient method for defect reduction planning, as it increases transparency by eliminating the need for black-box models.

Moreover, we study the statistical significance of the overlap scores and the F1-scores at the commit level. Similarly to RQ2, we use the Wilcoxon signed rank test. As shown in Table 8, the p-value is less than 0.05 in every case. This suggests that there is a statistically significant difference between the overlap performance of CounterACT and the other approaches. Moreover, using Wilcoxon signed rank tests across the F1-scores, CounterACT demonstrates statistical significance compared to the other planners having p-values consistently below the 0.05 threshold.

CounterACT produces more feasible plans than competing defect reduction approaches, as evidenced by the overlap score observed actions at commit -level (median overlap of 85.97% Table 8). In addition, CounterACT has the highest **Precision** and **Recall** trade-off among all the competing methods (average **F1-score** 72.83%).

7 DISCUSSION & IMPLICATIONS

The aim of this section is to discuss the implications of our approach by exploring the potential effectiveness and feasibility of our proposed planning approach.

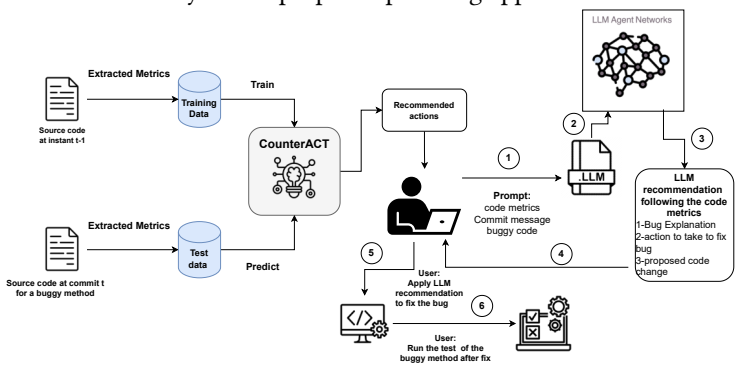

Fig. 10. Overview: Incorporate the LLM with CounterACT to generate actionable recommendations for developers

Feasibility of our plans for developers. Our research questions do not directly evaluate whether it is feasible for developers to apply CounterACT's plans. To remedy this limitation, we aim to provide some verification of their practical feasibility at the commit-level. To do so, we introduce an additional layer of support: the use of a LLMs (Large language models). Integrating an LLM with our generated plans aims to provide more actionable recommendations for developers.

To evaluate the efficacy of the LLM in aiding developers to act on CounterACT plans, we conduct experiments using CodeLlama [51]. In figure 10 we show the process of using LLM along with CounterACT to transform our generated plans into actions developers can implement.

We conduct experiments using the CodeLlama2 [51] LLM 34b trained by Meta. The model is fed the following information: the commit message, the code requiring a fix, and, importantly, the guidelines provided by CounterACT. These guidelines specifically articulate how the code metrics should be altered. In fact, our prompt followed a predefined template [49] (an example can be found in the reproduction package). Our experiments focused on the Kafka project, targeting 40 randomly selected bugs. For validation, we relied on the codeLlama2 model's guidance on code and ran test cases on the buggy method to ascertain whether the bugs had indeed been rectified. Our observations were as follows:

Effectiveness with guidance: When provided with our plans' guidance, the LLM generates actionable guidance for 40 randomly selected bugs. This guidance takes the form of bug explanations and a set of actions to take to fix the bug coupled with relevant code snippets in the illustrated example 1. In 28 out of the 40 bugs when following the LLM recommendation, the bug is fixed and the test cases pass. This highlights the potential of combining a LLM with strategic planning, leading to tangible and actionable solutions.

Effectiveness without guidance: When the vanilla LLM was only provided with the buggy code and commit message, its suggestions were less useful. Without guidance, the code snippets generated by the LLM lacked coherence and resulted in more failed test cases. Indeed, 26 out of 40 of these suggestions resulted in failed test cases. 14 of those failing cases were prevented when

using CounterACT plans, showing that our generated plans can guide the LLM in providing more useful suggestions.

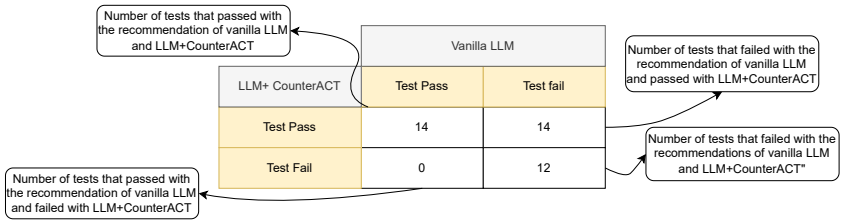

Fig. 11. Contingency table to Compare test outcomes for the vanilla LLM proposed bug fix VS the LLM with CounterACT guidance.

Moreover, we study the statistical significance of the LLM results, we use the McNemar statistical test [44] which aims at detecting differences between paired samples of two boolean variables. We organize the samples in a 2x2 contingency table as shown below in Figure 11. The null hypothesis of the McNemar test is that the distributions of unit tests passing with and without guidance are identical. Under this assumption, the off-diagonal terms in the table should be evenly distributed. Here, we see that no unit test passed without guidance and failed with guidance. Additionally, 14 unit tests passed with guidance and failed without guidance. It is worth noting that there are cases where the LLM often encounters difficulties despite CounterACT's guidance because the bug is specifically related to interactions with other classes or dependencies on different sections of the codebase. Since we do not provide such context to the LLM, no bug fix can be found in those instances. Despite a lack of context in certain cases, our results, when compared to LLM bug fixes without guidance, still yield a McNemar statistical test p-value of 0.000183 which is much less than 0.05. This provides evidence to reject the null hypothesis of no change, indicating that there is a statistically significant difference when using the LLM with guidance compared to not using it. While it is challenging to ascertain the exposure of LLMs to the project's fixing patches due to a lack of training details, we show that our enhancement to the LLM (through CounterACT) does indeed generally improve results since the vanilla LLM code edits failed the tests in 26 out of 40 experiments despite the risk of data leakage, all while demonstrating statistical significance.

The recommendations of LLMs are stochastic by design, different runs with the same prompt can induce different code fixes. This introduces an additional source of variability which may invalidate our previous observations. To address this issue, each inference of the LLM was run 10 times [3]. More specifically, for each unit test with/without CounterACT's guidance, we compute 10 LLM inferences and record the rates of success of the test. Since such rates are no longer boolean variables, the McNemar test is not applicable. Thus, we use a Repeated-Measure-ANOVA (RM-ANOVA) test [54]. The test yields a significant p-value of $1.0 \times 10^{-4}\%$.

8 THREATS TO VALIDITY

Internal Validity. Our study might be subject to potential alternative explanations for the observed outcomes. Although the use of alternative software metrics is not considered a direct threat, a pertinent concern arises when questioning the effectiveness of our chosen metrics in reducing defects. It is possible that our metrics are merely correlated with defect reduction, rather than being causally linked to it. This raises the possibility of other, unexplored factors or explanations that could be responsible for the observed decrease in defects. To mitigate this concern, we rely on metrics that have been widely used in prior works. While exploring the causal relationships of metrics is beyond the scope of this work, future work should involve examining additional metrics and factors. Moreover, the question remains whether the suggested plans can be credited for defect

```
1 // repository : apache/kafka
2 // commit id:c83719f3d8799ec35cf9af95fd93c97b910df570
3  public boolean hasRoomFor(long timestamp, ByteBuffer key, ByteBuffer value) {
4         if (isFull())
5             return false;
6         ...
7             - recordSize = DefaultRecord.sizeInBytes(nextOffsetDelta,
    timestampDelta, key,
8             value, Record.EMPTY_HEADERS);
9             + recordSize = DefaultRecord.sizeInBytes(nextOffsetDelta,
    timestampDelta, key, value, headers);
```

(a) Buggy code, the line with (-) represents the bug. The line with (+) represents the real bug fix from developers

```
1 //The bug is caused by the hasRoomFor method not taking into account the size of
     the record headers when determining whether there is enough room in the
    buffer for another record. As a result, it may allow records to be appended
    that exceed the maximum allowed size, leading to unexpected behavior or
    errors.
2 //To fix this issue, I would suggest changing the implementation as follows:
3 public boolean hasRoomFor(long timestamp, ByteBuffer key, ByteBuffer value,
    Header[] headers) {
4         if (isFull())
5             return false;
6             ....
7         recordSize = DefaultRecord.sizeInBytes(nextOffsetDelta, timestampDelta,
    key, value, headers);
8     }
9     return recordSize <= maxRecordSize;
```

(b) Proposed bug fix with the generated plan from CounterACT

```
1 //I would recommend adding some error handling or logging to show that the
    record could not be added due to lack of space. For example, you could add a
     log message like "Not enough space to add record" before returning null
2 public FutureRecordMetadata tryAppend(long timestamp, byte[] key, byte[] value,
    Header[] headers, Callback callback, long now) {
3     if (recordsBuilder.hasRoomFor(timestamp, key, value)) {
4         ...
5     } else {
6         logger.warn("Not enough space to add record");
7         return null;
```

(c) Proposed fix without the plan generated from CounterACT

Listing 1. The code snippet shown in (a) triggers a bug in the Kafka project at line 7. In (b) we show the bug fix recommendation of codeLlama when given the proposed plan "**NUMPAR:(0.995, 1.005)=>(1.025, 2.005) and don't change the rest metrics**". The LLM has provided good recommendations and fixed the method **hasRoomFor**. As in (c), we show the code snippet of the recommendation of the vanilla LLM. We see that the model ignores the bug and provides a logging statement to just track the bug. (You can find the prompt in our replication package.)

reduction or suppression. We believe that even if the plans do not guarantee defect reduction, they are correlated with defect reduction and can therefore provide a step toward improvement.

In our paper, the plans generated by CounterACT aim to identify factors commonly associated with improved software quality outcomes. For instance, a suggestion to "change the number of parameters" is for a file-level instance and the software metrics are specified to the object (e.g.,

class, method, interface). While this is still relatively general, it can prompt a review of methods that are overly complex or hard to maintain, guiding developers toward considering refactoring or redesigning for better modularity and readability. We recognize the importance of generating immediately actionable recommendations for practitioners. Thus we see value in how LLMs code edits were improved with CounterACT's guidance. We perceive this as a fruitful direction for future research to further refine the approach with fault-localization techniques and larger datasets.

External Validity. In our work, we focused on datasets consisting only of Apache projects. This can constrain the generalizability of our findings across diverse software development environments. However, we used Apache projects to maintain consistency with related work [27, 45]. Using the same context allows for direct comparison, thereby enhancing the validity and comparability of our findings. Future work should aim to extend the analysis to more diverse datasets to validate and expand our findings' applicability.

We implemented CounterACT on a selected set of software projects, which may limit the generalizability of our results to other datasets. To counter this limitation, we chose a range of real-world, open-source software applications for our analysis. Nevertheless, conducting further replication studies in alternative ecosystems would be beneficial for validating our findings.

While the release-level data employed states that it contains logs of past defects, we cannot guarantee the usefulness of overlap as an argument for claiming that we have captured specific bug fixes made by developers. Nonetheless, similarly to prior studies, overlap is an essential sanity check to ensure that the recommendations made by CounterACT are realistic. To complement overlap, we also measure the precision and recall. Finally, we also evaluate the approach at the commit level where labels are more specific to defects, thus minimizing confusion.

On the commit level, we use the public dataset provided by [64] which was processed using the base SZZ algorithm. We acknowledge that recent updates to the SZZ algorithm (e.g., RA-SZZ [40]) could very likely improve the results of our study, however by relying on an existing dataset to improve replicability and comparisons we were constrained to the choices made by that dataset. Future work could likely benefit from creating new datasets using more recent SZZ versions.

9 CONCLUSION

To the best of our knowledge, this study is the first to explore counterfactual-based planning using action rules mining for defect reduction. In this paper, we assess CounterACT, our novel approach for generating defect reduction plans, against state-of-the-art defect reduction planners on both release level and commit level. Results show that the plans are:

- Plausible: They are aligned with developers' actions in future releases or commits. CounterACT demonstrates better **Overlap**, **Precision**, **Recall**, and **F1-score** compared to competing approaches. Even at a finer granularity, the performance of our approach remains consistent. This shows that counterfactual analysis is effective at generating defect reduction plans and capturing relationships between code metrics even if the change is small.
- Actionable: Supporting `LLM` with CounterACT planning results in feasible recommendations. This has been evidenced by the fact that out of 40 bugs selected using this method, 28 have been successfully fixed and passed their respective test cases.

This research demonstrates that white-box approaches can provide effective defect-reduction plans that are competitive with state-of-the-art black-box approaches while offering a different view of what can be done in explainable software analytics.

10 DATA AVAILABILITY

Our data, and the scripts necessary to replicate our work is available, under an open license, using the following link: https://github.com/yukikh1234/counterACT_Defect_Reduction_Planning

REFERENCES

[1] Amritanshu Agrawal and Tim Menzies. 2017. "Better Data" is Better than "Better Data Miners" (Benefits of Tuning SMOTE for Defect Prediction). *CoRR* abs/1705.03697 (2017). arXiv:1705.03697 http://arxiv.org/abs/1705.03697

[2] Tiago L Alves, Christiaan Ypma, and Joost Visser. 2010. Deriving metric thresholds from benchmark data. In *2010 IEEE international conference on software maintenance*. IEEE, 1–10.

[3] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250. https://doi.org/10.1002/stvr.1486 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1486

[4] Barry Boehm and Victor Basili. 2001. Software Defect Reduction Top 10 List. *IEEE Computer* 34 (01 2001), 135–137. https://doi.org/10.1007/3-540-27662-9_26

[5] Nitesh Chawla, Kevin Bowyer, Lawrence Hall, and W. Kegelmeyer. 2002. SMOTE: Synthetic Minority Over-sampling Technique. *J. Artif. Intell. Res. (JAIR)* 16 (06 2002), 321–357. https://doi.org/10.1613/jair.953

[6] S.R. Chidamber and C.F. Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20, 6 (1994), 476–493. https://doi.org/10.1109/32.295895

[7] Cesar Couto, Pedro Pires, Marco Tulio Valente, Roberto S. Bigonha, and Nicolas Anquetil. 2014. Predicting software defects with causality tests. *Journal of Systems and Software* 93 (2014), 24–41. https://doi.org/10.1016/j.jss.2014.01.033

[8] Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. 2018. Explainable Software Analytics. *CoRR* abs/1802.00603 (2018). arXiv:1802.00603 http://arxiv.org/abs/1802.00603

[9] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2010. An extensive comparison of bug prediction approaches. *Proceedings - International Conference on Software Engineering*, 31–41. https://doi.org/10.1109/MSR.2010.5463279

[10] Agnieszka Dardzinska. 2012. *Action rules mining*. Vol. 468. Springer Berlin, Heidelberg.

[11] Usama M. Fayyad and Keki B. Irani. 1993. Multi-Interval Discretization of Continuous-Valued Attributes for Classification Learning. In *International Joint Conference on Artificial Intelligence*. https://api.semanticscholar.org/CorpusID:18718011

[12] Brooks Jr. Frederick P. 1995. *The Mythical Man-Month: Essays on Software Engineering*. Pearson Education.

[13] Bahar Gezici and Ayça Kolukisa Tarhan. 2022. Explainable AI for Software Defect Prediction with Gradient Boosting Classifier. In *2022 7th International Conference on Computer Science and Engineering (UBMK)*. 1–6. https://doi.org/10.1109/UBMK55850.2022.9919490

[14] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2012. A Systematic Literature Review on Fault Prediction Performance in Software Engineering. *IEEE Transactions on Software Engineering* 38, 6 (2012), 1276–1304. https://doi.org/10.1109/TSE.2011.103

[15] Steffen Herbold, Alexander Trautsch, and Fabian Trautsch. 2020. On the Feasibility of Automated Issue Type Prediction. *CoRR* abs/2003.05357 (2020). arXiv:2003.05357 https://arxiv.org/abs/2003.05357

[16] Jaitus Hihn and Tim Menzies. 2015. Data Mining Methods and Cost Estimation Models: Why is it So Hard to Infuse New Ideas?. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. 5–9. https://doi.org/10.1109/ASEW.2015.27

[17] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Hoa Dam, and John Grundy. 2020. An Empirical Study of Model-Agnostic Techniques for Defect Prediction Models. *IEEE Transactions on Software Engineering* PP (03 2020), 1–1. https://doi.org/10.1109/TSE.2020.2982385

[18] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Christoph Treude. 2018. AutoSpearman: Automatically Mitigating Correlated Metrics for Interpreting Defect Models. (06 2018).

[19] Viktor Jovanoski and Nada Lavrač. 2001. Classification Rule Learning with APRIORI-C. In *Progress in Artificial Intelligence*, Pavel Brazdil and Alípio Jorge (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 44–51.

[20] Marian Jureczko. 2011. Significance of Different Software Metrics in Defect Prediction. *Software Engineering: An International Journal* 1 (01 2011), 86–95.

[21] Marian Jureczko and Lech Madeyski. 2010. Towards Identifying Software Project Clusters with Regard to Defect Prediction *(PROMISE '10)*. Association for Computing Machinery, New York, NY, USA, Article 9, 10 pages. https://doi.org/10.1145/1868328.1868342

[22] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A Large-Scale Empirical Study of Just-in-Time Quality Assurance. *Software Engineering, IEEE Transactions on* 39 (06 2013), 757–773. https://doi.org/10.1109/TSE.2012.70

[23] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2013), 757–773. https://doi.org/10.1109/TSE.2012.70

[24] Vigdis Kampenes, Tore Dybå, Jo Hannay, and Dag Sjøberg. 2007. A systematic review of effect size in software engineering experiments. *Information and Software Technology* 49 (11 2007), 1073–1086. https://doi.org/10.1016/j.infsof.2007.02.015

[25] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. 2007. Predicting Faults from Cached History. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, USA, 489–498. https://doi.org/10.1109/ICSE.2007.66

[26] Rahul Krishna and Tim Menzies. 2017. From Prediction to Planning: Improving Software Quality with BELLTREE. *arXiv: Software Engineering* (2017).

[27] Rahul Krishna and Tim Menzies. 2020. From prediction to planning: Improving software quality with BELLTREE. *Empir. Softw. Eng.* 25 (2020), 3468–3500.

[28] Akshay Kumar and Rishabh Kumar. 2018. Uplift Modeling : Predicting incremental gains. https://api.semanticscholar.org/CorpusID:113397046

[29] Gichan Lee and Scott Uk-Jin Lee. 2023. An Empirical Comparison of Model-Agnostic Techniques for Defect Prediction Models. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 179–189. https://doi.org/10.1109/SANER56733.2023.00026

[30] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. 2016. Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning. *ArXiv* abs/1609.06570 (2016). https://api.semanticscholar.org/CorpusID:1426815

[31] Scott M. Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. *CoRR* abs/1705.07874 (2017). arXiv:1705.07874 http://arxiv.org/abs/1705.07874

[32] T.J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

[33] Thilo Mende and Rainer Koschke. 2010. Effort-Aware Defect Prediction Models. *European Conference on Software Maintenance and Reengineering*, 107–116. https://doi.org/10.1109/CSMR.2010.18

[34] Tim Menzies, Jeremy Greenwald, and Art Frank. 2007. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Trans. Software Eng.* 33 (01 2007), 2–13. https://doi.org/10.1109/TSE.2007.10

[35] Tim Menzies, Jeremy Greenwald, and Art Frank. 2007. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering* 33, 1 (2007), 2–13. https://doi.org/10.1109/TSE.2007.256941

[36] Mina Mobini and Sepideh Banihashemi. 2016. Analysis of Defect Prediction by using Object Oriented Metrics. (04 2016). https://doi.org/10.13140/RG.2.1.4075.8166

[37] Ramaravind K Mothilal, Amit Sharma, and Chenhao Tan. 2020. Explaining machine learning classifiers through diverse counterfactual explanations. In *Proceedings of the 2020 conference on fairness, accountability, and transparency*. 607–617.

[38] Nachiappan Nagappan and Thomas Ball. 2007. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. *Proceedings - 1st International Symposium on Empirical Software Engineering and Measurement, ESEM 2007*, 364–373. https://doi.org/10.1109/ESEM.2007.13

[39] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. 2010. Change Bursts as Defect Predictors. *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*. https://doi.org/10.1109/ISSRE.2010.25

[40] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. 2018. 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)*. 380–390. https://doi.org/10.1109/SANER.2018.8330225

[41] Paloma Oliveira, Marco Tulio Valente, and Fernando Paim Lima. 2014. Extracting relative thresholds for source code metrics. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 254–263.

[42] Axel Parmentier and Thibaut Vidal. 2021. Optimal counterfactual explanations in tree ensembles. In *International Conference on Machine Learning*. PMLR, 8422–8431.

[43] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2019. Fine-grained just-in-time defect prediction. *Journal of Systems and Software* 150 (2019), 22–36. https://doi.org/10.1016/j.jss.2018.12.001

[44] Matilda Q. R. Pembury Smith and Graeme D. Ruxton. 2020. Effective use of the McNemar test. *Behavioral Ecology and Sociobiology* 74, 11 (2020), 133. https://doi.org/10.1007/s00265-020-02916-y

[45] Kewen Peng and Tim Menzies. 2021. Defect reduction planning (using timelime). *IEEE Transactions on Software Engineering* 48, 7 (2021), 2510–2525.

[46] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2021. JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction. *CoRR* abs/2103.07068 (2021). arXiv:2103.07068 https://arxiv.org/abs/2103.07068

[47] Chanathip Pornprasit, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Michael Fu, and Patanamon Thongtanunam. 2021. PyExplainer: Explaining the Predictions of Just-In-Time Defect Models. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 407–418. https://doi.org/10.1109/ASE51524.2021.9678763

[48] Dilini Rajapaksha, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Christoph Bergmeir, John Grundy, and Wray Buntine. 2021. SQAPlanner: Generating Data-Informed Software Quality Improvement Plans. *IEEE Transactions on Software Engineering* PP (04 2021), 1–1. https://doi.org/10.1109/TSE.2021.3070559

[49] Facebook Research. 2023. llama-recipes. https://github.com/facebookresearch/llama-recipes/tree/main Accessed: 2023-06-28.
[50] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. " Why should i trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 1135–1144.
[51] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]
[52] Cynthia Rudin. 2019. Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead. arXiv:1811.10154 [stat.ML]
[53] Chris Russell. 2019. Efficient search for diverse coherent explanations. In *Proceedings of the Conference on Fairness, Accountability, and Transparency*. 20–28.
[54] Andrew Rutherford. 2011. *ANOVA and ANCOVA: a GLM approach*. John Wiley & Sons.
[55] Raed Shatnawi. 2010. A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems. *IEEE Transactions on software engineering* 36, 2 (2010), 216–225.
[56] Jiho Shin, Reem Aleithan, Jaechang Nam, Junjie Wang, and Song Wang. 2021. Explainable Software Defect Prediction: Are We There Yet? arXiv:2111.10901 [cs.SE]
[57] Jiho Shin, Reem Aleithan, Jaechang Nam, Junjie Wang, and Song Wang. 2021. Explainable Software Defect Prediction: Are We There Yet? *CoRR* abs/2111.10901 (2021). arXiv:2111.10901 https://arxiv.org/abs/2111.10901
[58] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM Sigsoft Software Engineering Notes* 30. https://doi.org/10.1145/1082983.1083147
[59] Lukas Sykora and Tomáš Kliegr. 2020. Action Rules: Counterfactual Explanations in Python. In *RuleML+RR*.
[60] Chakkrit Tantithamthavorn, Ahmed E. Hassan, and Kenichi Matsumoto. 2020. The Impact of Class Rebalancing Techniques on the Performance and Interpretation of Defect Prediction Models. *IEEE Transactions on Software Engineering* 46, 11 (2020), 1200–1219. https://doi.org/10.1109/TSE.2018.2876537
[61] Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, and John Grundy. 2021. Actionable analytics: Stop telling me what it is; please tell me what to do. *IEEE Software* 38, 4 (2021), 115–120.
[62] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2019. The Impact of Automated Parameter Optimization on Defect Prediction Models. *IEEE Transactions on Software Engineering* 45, 7 (2019), 683–711. https://doi.org/10.1109/TSE.2018.2794977
[63] Alexander Trautsch, Johannes Erbel, Steffen Herbold, and Jens Grabowski. 2021. On the differences between quality increasing and other changes in open source Java projects. *CoRR* abs/2109.03544 (2021). arXiv:2109.03544 https://arxiv.org/abs/2109.03544
[64] Alexander Trautsch and Steffen Herbold. 2021. *PROMISE 2021 Defect Prediction Challenge Data*. https://doi.org/10.5281/zenodo.8370181
[65] Sandra Wachter, Brent Mittelstadt, and Chris Russell. 2017. Counterfactual explanations without opening the black box: Automated decisions and the GDPR. *Harv. JL & Tech.* 31 (2017), 841.
[66] Romi Wahono. 2015. A Systematic Literature Review of Software Defect Prediction: Research Trends, Datasets, Methods and Frameworks. *Journal of Software Engineering* 1 (05 2015).
[67] Robert F Woolson. 2007. Wilcoxon signed-rank test. *Wiley encyclopedia of clinical trials* (2007), 1–3.
[68] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. 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* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 157–168. https://doi.org/10.1145/2950290.2950353

Received 2023-09-28; accepted 2024-04-16