A Method for Reducing Defect Introduction Based on Machine Learning Considering Time-Series Code Changes

2410064 Hiroto Sasagawa

In modern software engineering, development processes have transitioned from traditional waterfall models to agile and continuous integration/ continuous delivery (CI/CD) frameworks. This shift allows for rapid responses to market needs. However, it also requires frequent code changes. Continuous modifications often compromise the consistency between new and existing code. Consequently, these changes increase the risk of introducing software defects. The cost of fixing such defects increases exponentially as they move toward later stages of production. Therefore, identifying potential defects early is a critical challenge for quality assurance.

Historically, defect prediction research has been based on structural metrics, such as cyclomatic complexity and lines of code (LOC). These metrics analyze the state of the code at a specific point in time. However, structural metrics cannot capture the dynamic characteristics of the development process. Recent studies have begun to utilize change metrics derived from version control systems (VCS). These approaches show a stronger correlation with defect occurrences than structural metrics. Nevertheless, existing change-related methods face two issues. First, they do not consider the irregular timing of code commits. Second, they focus on a single component, such as methods or commits, without combining local and global perspectives.

This work investigates a defect prediction method that incorporates three approaches to address these issues. First, the method treats commits as irregularly occurring events. Rather than measuring data at fixed intervals, it focuses on the differences between consecutive commits to reflect the actual changes of software development. This enables the machine learning model to track changes in complexity that may not be apparent in structural metrics.

Second, the proposed method combines multi-level metrics to capture both local and global trends. We extract per-method metrics to reflect local implementation errors. These include changes in lines of code, token counts, and cyclomatic complexity. Simultaneously, we extract per-commit metrics to represent global impact.These include the number of modified files, lines added or deleted, and "change scattering" measured by entropy. By combining these perspectives, the machine learning model can identify risks where a minor local change might have a significant global impact.

Third, we introduce a review prioritization method based on "effort-aware" defect prediction. In real-world development, resources for code review are limited. It is often impossible to review every single change in detail.

conventional methods often assume that review effort is uniform across all components. However, the actual effort depends on the scale and complexity of the changes. Our method defines a corrected review effort using the scale and complexity of the changes. Next, we apply combinatorial optimization algorithms to detect more defects with less effort.

To validate the effectiveness of our approach, we conducted experiments on five open-source projects: Elasticsearch, Hazelcast, Netty, OrientDB, and Neo4j . These projects cover diverse domains, including distributed search engines, network frameworks, and multi-model databases, ensuring that the findings are applicable to various software architectures. We used the BugHunter dataset, which provides information about defect-introducing and defect-fixing commits. The defect prediction model was built using a random forest, a machine learning algorithm capable of handling nonlinear relationships and providing feature importance. Performance was evaluated using F1 scores and Area Under the Curve (AUC) through 10-fold cross-validation.

We confirmed that the proposed method consistently improves the accuracy of the prediction in all target projects. Compared to conventional models using only structural metrics, the inclusion of change-related features led to an increase in the F1 score. In particular, for projects where the prediction performance was low using conventional methods, the proposed approach significantly improved the prediction performance. The AUC values are high for all projects, and this reflects a strong ability to distinguish between defect-introducing commits and defect-fixing commits. Statistical analysis using the McNemar test indicated that these performance gains were consistent and not attributable to random variation.

Furthermore, we found that the review prioritization method is effective under actual resource constraints. Even with a little review effort, the proposed method was able to identify a large majority of defects. For a given effort, the proposed method detected more defects than conventional methods, suggesting that this model effectively prioritizes changes with a high risk of defect introduction. The Wilcoxon signed-rank test supported the significance of this improvement. The proposed prioritization method achieved an efficiency gain, identifying approximately 70 to 75% of all defects within only 20% of the total review effort. These findings suggest that developers can find most potential issues by their a little review effort.

Analysis of feature importance revealed that commit-level metrics, such as the number of added lines and modified files, served as the strongest predictors of defect risk in most projects. However, method-level metrics also provided critical information; for example, the change in token counts was identified as a significant predictor in the Netty project. Interestingly, Partial

Dependence Plot (PDP) analysis showed a trend where smaller changes often had a higher probability of containing defects. This suggests that while large commits indicate broad impact, subtle modifications at the method level can effectively signal defect risks. Additionally, small changes are easier to review, meaning defects within them are more likely to be identified and recorded in the dataset.

In conclusion, this research demonstrates that using the time-series data of code changes and combining multi-level metrics improve the defect prediction performance. By providing a model that considers the review effort, we offer a practical decision-support tool for developers. Future work should focus on identifying the specific intent behind changes, such as whether a change is for a new feature or refactoring. Understanding the purpose of the change makes it easier to detect the cause of defects and helps create more targeted advice.