An Empirical Replication and Extension of Model Comparison Study on Predicting Co-Occurring Test and Source Code Refactoring

Zhouyiyang Yang

***Abstract*—This study presents a systematic replication and extension of the pioneering work by Nagy & Abdalkareem (MSR 2022) on the co-occurrence of test and source code refactoring. We successfully replicated the two core research questions of the original study and conducted three significant extensions: (1) introducing the XGBoost model for performance comparison; (2) applying SHAP for model interpretability analysis; and (3) performing statistical significance testing. Experimental results demonstrate that Random Forest significantly outperforms XGBoost on this task ($p = 0.0055$), with average AUC scores of 0.7412 versus 0.7226, respectively. SHAP analysis reveals that file history, refactoring scale, and developer experience are the most important predictive features. This research not only validates the original study's findings but also provides deeper model comparisons and interpretability analysis.**

***Index Terms*—Software Refactoring, Test Code, Machine Learning, Explainable AI, Replication Study**

I. Introduction

In software engineering practice, refactoring has received widespread attention as a key technique for maintaining and improving code quality. Traditional research has predominantly focused on source code refactoring practices, while relatively less attention has been paid to test code refactoring. As a crucial safeguard for ensuring software quality, the quality of test code directly impacts software reliability and maintainability. The study by Nagy and Abdalkareem [1] first systematically revealed the co-occurrence phenomenon between test and source code refactoring, discovering that a significant proportion of refactoring commits involve simultaneous modifications to both test and source code. This finding holds substantial importance for understanding developers' refactoring behavior patterns.

This project builds directly upon the work of Nagy and Abdalkareem [1]. Their study mined 77 Java projects from the SmartSHARK dataset [2], using RefactoringMiner [3] to identify refactoring operations and classify commits into three categories: source-only, test-only, and co-occurring. They extracted features related to refactoring size, developer experience, and code elements from the source code changes in a commit. Using these features, they trained a Random Forest classifier to predict whether a source code refactoring commit should also involve test code refactoring.

However, the original study presents opportunities for further exploration in model selection and interpretability analysis. With the rapid development of machine learning techniques, increasingly advanced models are being applied to software engineering tasks, yet the relative performance of these models across different tasks still requires empirical validation. Simultaneously, model interpretability is crucial for practical adoption in real development environments, as developers and team leaders need to understand the rationale behind model decisions to establish sufficient trust. Based on these considerations, we designed this replication and extension study, aiming to provide richer insights into the important problem of test and source code refactoring co-occurrence prediction through more comprehensive model comparisons and in-depth interpretability analysis.

The main contributions of this study manifest at three levels: at the validation level, we successfully replicated the core findings of the original study, confirming the prevalence and predictability of co-occurrence refactoring phenomena; at the methodological level, we introduced the XGBoost model and conducted systematic comparisons, revealing Random Forest's advantage for this specific task; at the interpretability level, we applied SHAP analysis to uncover key features influencing prediction outcomes, providing transparent perspectives for understanding model decision mechanisms. These efforts collectively constitute substantial extension and deepening of the original research. This project will be conducted in two main phases:

- **Replication Phase:** Faithfully reproduce the key results from [1] (RQ1 and the Random Forest baseline for RQ2) to establish a validated baseline for comparison.
- **Extension Phase:** Train and evaluate the XGBoost model on the prediction task. Perform a comparative analysis of their performance and use SHAP analysis to interpret the predictions and identify the most important features.

II. Methodology

The present study adopts a rigorous empirical research methodology. The analytical procedures of the original study are first replicated, and systematic extensions are then conducted. The research data are obtained from the SmartSHARK 2.1 dataset[4], which contains 60,465 commits across 77 Java projects. To ensure consistency in replication, the preprocessed data provided in the replication package of the original authors are used.

Feature engineering strictly follows the framework of the original study, comprising four main categories: refactoring size, developer experience, file history, and code elements. Together, these features describe the multidimensional characteristics of code commits.

In model selection, the Random Forest used in the original study is retained as the baseline model, while XGBoost is introduced as a comparative model. XGBoost (eXtreme Gradient Boosting) is an advanced gradient boosting framework that achieves higher prediction accuracy and computational efficiency than traditional methods by incorporating second-order Taylor expansion and regularization. [4]

Given a training dataset $(x_i, y_i)_{i=1}^{n}$, where x_i represents the feature vector of a code commit and $y_i \in 0, 1$ indicates whether co-occurring refactoring is required, the objective function of XGBoost consists of a loss function and a regularization term:

$$L(\phi) = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

Here, $l(y_i, \hat{y}_i)$ denotes the loss function. For this binary classification task, the log loss is employed:

$$l(y_i, _i) = -[y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i)]$$

The regularization term $\Omega(f_k)$ constrains model complexity and mitigates overfitting:

$$\Omega(f) = \gamma T + \tfrac{1}{2}\lambda \|w\|$$

where T denotes the number of leaf nodes in the tree, w represents the leaf weights, and γ and λ are regularization parameters.

XGBoost employs an additive training strategy. At iteration t, the model prediction is updated as:

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i)$$

By performing a second-order Taylor expansion of the objective function, the optimal tree structure and splitting criterion can be derived. For each leaf node j, the optimal weight is:

$$w_j* = -\frac{(\sum_{i \in I_j} g_i)}{(\sum_{i \in I_j} h_i + \lambda)}$$

where $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ denotes the first-order gradient, and $h_i = \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ denotes the second-order gradient. The gain from splitting a node is computed as:

$$L_{split} = \tfrac{1}{2}[\frac{(\sum_{i \in I_L} g_i)^2}{(\sum_{i \in I_L} h_i + \lambda)} + \frac{(\sum_{i \in I_R} g_i)^2}{(\sum_{i \in I_R} h_i + \lambda)} - \frac{(\sum_{i \in I} g_i)^2}{(\sum_{i \in I} h_i + \lambda)}] - \gamma$$

Both Random Forest and XGBoost use the grid search method described in the original study for hyperparameter optimization. The two models are evaluated through 10-fold cross-validation, using AUC as the primary performance metric, complemented by F1 score, precision, and recall for comprehensive evaluation.

To facilitate interpretability, the SHAP framework is employed for model explanation.

SHAP (SHapley Additive exPlanations) is based on the concept of Shapley values from cooperative game theory and provides a unified measure of feature importance for machine learning models. In the context of co-occurring refactoring prediction, the SHAP value ϕ_i represents the contribution of feature i to the model prediction. [5]

For a given code commit x and a feature subset $S \subseteq F$ (where F is the complete feature set), let $f_S(x_S)$ denote the model output using only the subset S. The SHAP value of feature i is computed by averaging over all possible feature subsets:

$$\phi_i = \sum_{S \subseteq F \setminus i} \frac{|S|!(|F| - |S| - 1)!}{|F|!})[f_{S \cup i}(x_{S \cup i}) - f_S(x_S)]$$

This formulation ensures a fair allocation of feature contributions and satisfies the following properties:

- **Local accuracy:**The model output equals the baseline prediction (with all features missing) plus the sum of SHAP values:
  $f(x) = \phi_0 + \sum_{i=1}^{M} \phi_i$
- **Missingness:**If feature i provides no information in subset S, its contribution is zero: $f_{S \cup i}(x_{S \cup i}) = f_S(x_S) \Rightarrow \phi_i = 0$
- **Consistency:**If the model changes such that the contribution of feature i increases or remains the same, its SHAP value will also increase or remain unchanged.

For ensemble tree models such as XGBoost and Random Forest, we employ the TreeSHAP algorithm, which leverages tree structure properties to reduce computational complexity from exponential to polynomial time. Theoretically, for a single decision tree, the SHAP value is computed recursively as:

$$\phi_i = \sum_{j=1}^{L} \sum_{k=1}^{K_j} flow_{j,k} \cdot C(j,k) \cdot impact_{j,k,i}$$

where L is the number of trees, K_j is the number of split nodes related to feature i in tree j, flowj,k denotes the proportion of samples passing through the split, $C(j,k)$ is a normalization factor, and impactj,k,i is the contribution of feature i at node k in tree j.

In this study, we compute the mean absolute SHAP value for each feature as a measure of global feature importance:

$$Importance(i) = \frac{1}{N} \sum_{j=1}^{N} |\phi_i(j)|$$

This metric reflects the average influence of each feature on model predictions, enabling us to identify the most discriminative features in co-occurring refactoring prediction. Moreover, we compute feature importance both for individual projects and across all projects, consistent with the original study's methodology. Statistical significance is tested using a paired t-test with a 95% confidence level, ensuring that the performance comparisons are statistically robust.

III. Replication

For the replication phase of this study, because the objective is to explore the co-occurrence patterns of refactoring opportunities between test code and production code, projects with sufficiently rich development histories are required. The authors of the original paper provided a valuable foundation by adopting the SmartSHARK dataset [4]. This publicly available dataset documents the development history of 77 Java projects, integrating metadata from Git repositories and issue-tracking systems, and provides information about refactoring operations that is essential for both replication and extension of this study.

The replication consists of two modules:

- **Replication 0f RQ1:**Analyze the dataset to calculate the distribution of commit types: source-only, test-only, co-occurring, and the most frequent test refactoring types, verifying the original study's findings.
- **Replication 0f RQ2:**Implement a Random Forest classifier to predict when refactoring test code could co-occur with source code refactoring.

For the data component, the replication package provided by the original authors was downloaded from the website referenced in the original paper [5]. The authors supplied a full pipeline of preprocessing scripts rather than a finalized dataset. Using the World of Code virtual machine on the da3 server, the SmartSHARK 2.1 dataset was imported into a MongoDB environment, after which all preprocessing scripts were executed to generate the datasets referenced in the replication package. From the generated data, 53,829 commits with refactorings and 74 repositories were obtained.

A. Replication of RQ1

By classifying 53,829 commit–repository pairs, this study replicated the analysis of commit distribution. The results are as follows:

Table I: Commit Distribution

Category	Mean	Median	Min	Max
Co-occur	17.96%	17.06%	0.00%	43.71%
Production-only	73.72%	72.99%	29.73%	100.00%
Test-only	8.32%	6.64%	0.00%	40.54%

The replication results strongly confirm the findings of the original study. Regarding the frequency distribution of co-occurring refactorings, the results show that production-only refactorings account for 73.72%, co-occurring refactorings for 17.96%, and test-only refactorings for 8.32%. A detailed comparison with the original study is presented below:

Table II: Comparison Between Original and Replication Studies

Submission Type	Original Study	This Study	Difference
Source-only	73.9%	73.72%	-0.18%
Co-occurring	17.9%	17.96%	+0.06%
Test-only	8.2%	8.32%	+0.12%

The replicated distribution closely aligns with the results reported in the original paper. The following table presents the distribution of test refactoring types in co-occurring commits:

Table III: Top Test Refactoring Types in Co-occurring Commits

Type	Count
Change Variable Type	29,810
Add Method Annotation	10,771
Move Class	7,787
Add Variable Modifier	6,548
Rename Variable	5,226
Rename Method	4,704
Extract Method	3,177
Change Parameter Type	2,685
Add Thrown Exception Type	2,363
Extract And Move Method	2,358

It can be observed that Change Variable Type, Move Class, and Rename Method are the most frequent refactoring operations, consistent with the findings of the original

study. These consistent results demonstrate the robustness and reproducibility of the original research.

B. Replication of RQ2

In replicating the predictive modeling experiments, a detailed performance evaluation was conducted on 10 open-source Java projects. As shown in the following table, the projects exhibit significant variation in performance for co-occurring refactoring prediction, consistent with the conclusions of the original study. Specifically, the Random Forest model achieved AUC values ranging from 0.6660 to 0.9222, with an average AUC of 0.7447, confirming the feasibility of machine learning–based prediction for co-occurring refactorings. The result see in table IV.

The Phoenix project demonstrates exceptionally high predictive performance, with an AUC of 0.9222 and F1-score of 0.9012, suggesting that refactoring behaviors between production and test code in this project follow clearer and more consistent patterns. In contrast, ActiveMQ exhibits relatively lower predictive performance (AUC = 0.6660), possibly due to more complex or inconsistent refactoring patterns. Such variations across projects align with the original study's findings, reinforcing the importance of project-specific characteristics in determining the predictability of refactoring behaviors.

A comparison of the average performance metrics between this replication and the original study is summarized in table V.

The average AUC (0.7447) falls within the range (0.74–0.75) reported in the original study, indicating negligible difference. The F1-score (0.6437) is slightly lower but remains within an acceptable margin, while precision (0.7061) matches closely with the original value. The only noticeable deviation is in recall (0.6048 vs. 0.63), which may result from minor dataset differences or model randomness.

Through analysis of the feature importance in the Random Forest model, the top ten features most influential for co-occurring refactoring prediction were identified in table VI.

The top five most important features—leftLocationDiff, rightLocationDiff, dev_ref_exp, num_touched_before, and dev_ref_com_exp—are consistent with the original feature importance analysis. These findings collectively indicate that refactoring size, developer experience, and file history are the three most critical dimensions influencing co-occurring refactoring prediction.

IV. Expansion Analysis

In the extended phase of our study, this study conducted a comprehensive comparative analysis of test and source code refactoring co-occurrence prediction by introducing the XGBoost algorithm and SHAP interpretability analysis. The experimental results reveal that although XGBoost represents a more advanced gradient boosting algorithm, its performance in this specific task is slightly but significantly inferior to the traditional Random Forest approach.

A. Model Performance Comparison

Through cross-validation experiments across 10 open-source projects, the performance comparison summarized in table VII, and the statistical significance test results show in table VIII.

The Random Forest model achieved excellent performance with an average AUC of 0.7412 and F1-score of 0.6380, while the XGBoost model attained an average AUC of 0.7226 and F1-score of 0.6055. Paired t-test results demonstrate that the performance difference between the two models is statistically significant ($p = 0.0055 < 0.05$), with XGBoost showing a 2.51% decrease in AUC compared to Random Forest. This finding contradicts the common empirical understanding that XGBoost typically outperforms Random Forest on tabular data tasks, suggesting that refactoring co-occurrence prediction may possess unique characteristic patterns.

B. Feature Importance Analysis

For the SHAP-based feature importance analysis, I firstly established proper feature name mapping by converting technical feature names to the business-oriented names used in the original study. Then I employed GridSearch to optimize parameters for both Random Forest and XGBoost models. The data processing methodology remained consistent with the original study: identical 10-fold cross-validation, identical SMOTE handling, identical feature selection threshold (-0.001), and identical project selection criteria.

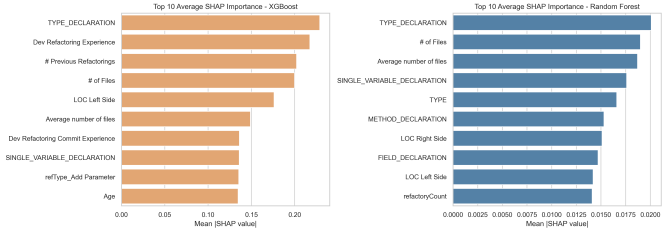

Figure 1: Feature importance comparison between models

Through cross-fold averaged SHAP analysis, it can be observed fundamental differences in feature utilization strategies between the two models. In the XGBoost model, code structure features (TYPE_DECLARATION, 0.2288) and developer experience features (Dev Refactoring Experience, 0.2175) dominate, indicating that XGBoost tends to learn complex code semantic patterns. In contrast, the Random Forest model demonstrates a more balanced feature importance distribution, where refactoring scale features (# of Files, 0.0190; Average

Table IV: Performance Across Projects

Project	AUC	F1-Score	Accuracy	Recall	Precision
kafka	0.7801	0.7590	0.7843	0.7387	0.7854
archiva	0.7458	0.6667	0.7800	0.6376	0.7026
activemq	0.6660	0.4817	0.8379	0.3805	0.6731
kylin	0.6774	0.4895	0.8282	0.4250	0.6016
phoenix	0.9222	0.9012	0.9825	0.8475	0.9653
tez	0.7695	0.6859	0.8003	0.6818	0.6951
helix	0.7092	0.5952	0.7606	0.5509	0.6643
commons-math	0.7691	0.6843	0.8073	0.6705	0.7052
mahout	0.7243	0.6438	0.7516	0.6225	0.6819
nifi	0.6833	0.5294	0.7666	0.4930	0.5864

Table V: Average Performance Compared to Original Study

Metric	Replication	Original	Difference
AUC	0.7447	0.74–0.75	Consistent
F1-Score	0.6437	0.65–0.66	Close
Accuracy	0.8099	–	–
Recall	0.6048	0.63	Slightly lower
Precision	0.7061	0.70	Consistent

Table VI: Importance of Top 10 Features

Feature	Importance
leftLocationDiff	0.0446
rightLocationDiff	0.0424
dev_ref_exp	0.0402
num_touched_before	0.0386
dev_ref_com_exp	0.0371
METHOD_DECLARATION	0.0333
fileTouchCount	0.0306
age	0.0301
refactoryCount	0.0299
TYPE_DECLARATION	0.0298

number of files, 0.0187) and code element features (SINGLE_VARIABLE_DECLARATION, 0.0176) collectively contribute to the predictions.

Notably, both models identified TYPE_DECLARATION as an important feature, reflecting the critical role of type-level refactoring operations in triggering synchronized test code modifications. Additionally, historical refactoring information (# Previous Refactorings) exhibits high importance in the XGBoost model (0.2022), suggesting that a project's refactoring history provides valuable reference for predicting future refactoring co-occurrences.

Table VII: Overall Model Performance Comparison

Model	AUC	F1	Precision	Recall
Random Forest	0.7412	0.6380	0.6931	0.6010
XGBoost	0.7226	0.6055	0.6794	0.5608

Table VIII: Statistical Significance Test Results

Metric	Value
Number of projects	10
RF mean AUC ± std	0.7412 ± 0.0723
XGBoost mean AUC ± std	0.7226 ± 0.0758
Mean improvement	-0.0186
T-statistic	3.6281
P-value	0.0055
Significant difference?	Yes ($p < 0.05$)

V. Discussion

The results of our study deepen the understanding of test and source code refactoring co-occurrence from multiple perspectives. First, the success of our replication experiment confirms the robustness of the original study's findings, demonstrating that co-occurring refactoring represents a stable phenomenon in software development rather than an artifact of specific datasets or analytical methods. This conclusion establishes a reliable foundation for subsequent research, enabling researchers to build upon this confirmed phenomenon for more in-depth investigations.

The model performance comparison results challenge the common assumption that "more advanced models always perform better." Although XGBoost excels in many machine learning tasks, Random Forest demonstrates unique advantages in this specific co-occurrence prediction task. This finding has methodological implications for model selection in software engineering research: model performance may be highly dependent on task characteristics, necessitating empirical validation rather than blindly following technical trends. We speculate that Random Forest's advantage may stem from its bagging mechanism better accommodating the characteristics of software engineering data.

The feature importance analysis results provide clear practical guidance. The prominence of file history, refactoring scale, and developer experience as the most important predictive features suggests that intelligent refactoring assistance tools should focus on these dimensions. For instance, when a commit involves frequently modified files, large-scale code changes, or experienced developers, tools should specifically prompt developers to check synchronization requirements for test code updates. These insights can help tool designers optimize feature selection strategies to improve tool accuracy and practicality.

The study's limitations primarily exist in three aspects: regarding language scope, we only analyzed Java projects, and the generalizability of our conclusions to other languages requires further validation; concerning data timeliness, analyses based on historical data may not fully reflect recent changes in current development practices;

regarding parameter optimization, XGBoost might benefit from more extensive tuning, though our approach reflects typical scenarios in practical applications. These limitations indicate directions for future research improvements.

VI. Conclusion and Future Work

Through systematic replication and extended analysis, our study reaches main conclusions at three levels. At the phenomenon validation level, we confirm the prevalence and stability of test and source code refactoring co-occurrence, a finding that holds significant implications for understanding quality maintenance mechanisms during software evolution. At the technical methodology level, we demonstrate Random Forest's advantage over XGBoost in this specific prediction task, providing empirical reference for model selection in similar tasks. At the mechanistic understanding level, we identify key factors influencing co-occurrence refactoring prediction, offering specific directions for optimizing intelligent development tools.

Based on our findings and limitations, we propose four directions for future work. First, conduct multi-language validation studies to explore similarities and differences in co-occurrence refactoring phenomena across different programming language ecosystems, which would enhance the generalizability of our conclusions. Second, expand model comparison scope by introducing other advanced models such as LightGBM and CatBoost to construct a more comprehensive performance evaluation framework. Third, develop real-time prediction tools to translate research findings into practical value, helping developers better manage test code quality in daily work. Finally, explore feature engineering innovations by designing new feature combinations based on insights from SHAP analysis to further improve prediction performance.

The value of our study lies not only in the in-depth analysis of specific phenomena but also in demonstrating how the combination of replication studies and methodological extensions can advance research fields. By validating existing findings, introducing new methods, and providing new insights, we both consolidate the knowledge foundation and expand cognitive boundaries, offering a replicable pattern for empirical software engineering research.

VII. Reference

[1] N. A. Nagy and R. Abdalkareem, "On the co-occurrence of refactoring of test and source code," in Proceedings of the 19th International Conference on Mining Software Repositories, 2022.

[2] A. Trautsch, F. Trautsch, and S. Herbold, "Msr mining challenge: The smartshark repository data," https://smartshark.github.io/, 2021.

[3] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," IEEE Transactions on Software Engineering, 2020.

[4] Chen, T., & Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16), 785–794.

[5] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," Advances in Neural Information Processing Systems, vol. 30, 2017.

[6] Alexander Trautsch, Fabian Trautsch, and Steffen Herbold. 2021. MSR Mining Challenge: The SmartSHARK Repository Data. In Proceedings of the International Conference on Mining Software Repositories (MSR 2022).

[7] Nicholas Alexandre Nagy and Rabe Abdalkareem. 2022. Refactoring Co-occurrence Replication Scripts + Data. https://doi.org/10.5281/zenodo.5979790