S1: Hi everyone, today I'm going to present /pri'zent/ my work, which is an empirical replication and extension of study on code refactoring.
S2: About the original paper, we know the software requiring continuous refactoring to keep its quality and health. Source code is a well-studied topic, while test code refactoring is often overlooked.
Akey finding from the original paper by Nagy and Abdalkareem in 2022revealed that: on average, 17.9% of refactoring commits are "co-occurring," meaning they involve changes to both test and source code. and this is more than double the rate of commits that only refactor test code.
This motivates us that: by understanding and predicting this co-occurrence, we can build smarter developer tools that proactively suggest when test code needs to be refactored alongside source code.
S3: The original study established a strong foundation. It had two main research questions: RQ1 was an empirical study on the frequency and types of co-occurring refactorings. RQ2 used a Random Forest model to predict this co-occurrence, they found the features of source code refactoring can effectively predict whether it is necessary to refactor the test code simultaneously.
However, this work had some limitations. First, it only explored one model – the Random Forest. Second, it provided limited interpretability /ɪnˌtɜːrprɪtəˈbɪlɪti/ into why the model made its predictions.
S4: Therefore, my research is guided by the following questions. First, the Replication RQ1: Can we replicate the original study's findings on co-occurring refactorings? Second, the Replication and Extension RQ2: Can we replicate the Random Forest baseline? And then, our key extensions: Can XGBoost, a more advanced algorithm, outperform Random Forest? And can SHAP analysis provide better insights into key predictive features compared to the original Gini importance analysis?
S5: To achieve this, I structured my work into two main phases. Phase 1 was Replication, which involved data loading, preprocessing, and replicating both the descriptive statistics of RQ1 and the Random Forest baseline for RQ2. Phase 2 was Extension, where I trained and compared XGBoost against Random Forest, and conducted a SHAP analysis for feature importance.
S6: about the key technologies in the extension. XGBoost is an efficient and powerful gradient boosting algorithm. It often demonstrates strong performance on structured data. The key difference is that while Random Forest uses "bagging" to build trees in parallel, XGBoost uses "boosting," which builds trees sequentially, with each new tree correcting the errors of the previous ones, often leading to higher accuracy. For interpretability, SHAP is a game theory-based method that provides a unified and theoretically sound measure of feature importance for any model, explaining its individual predictions.
S7: So these are the tools and core libraries. and For data, I strictly followed the original study by using the SmartSHARK 2.1 dataset. I obtained the original authors' replication package, which contained preprocessing scripts that can run to generate the final dataset for the analysis.

S8: I evaluated these models using standard metrics, with AUC as the primary metric, complemented by F1-Score, Precision, and Recall. Of course, 10-fold cross-validation is used to ensure the robustness of our results. The success criteria were threefold: First, a successful replication, meaning similar RQ1 statistics and Random Forest performance. Second, a clear model comparison outcome, whether XGBoost improved or decreased performance significantly. And third, gaining deeper insights through SHAP analysis.
S9: This is the result of RQ1 replication. found that source-only refactorings account for 73.72%, co-occurring refactorings for 17.96%, and test-only refactorings for 8.32%. This distribution is nearly identical to the original study's findings. Furthermore, the most frequent test refactoring types in co-occurring commits were "Change Variable Type," "Move Class," and "Rename Method," which also perfectly matches the original results.
S10: This is the result of RQ2 replication. RF model achieved an average AUC of 0.7447 across 10 projects, which falls perfectly within the original study's reported range of 0.74 to 0.75. Performance varied by project, with Phoenix showing exceptional predictability and ActiveMQ being more challenging, mirroring the original findings. Other metrics like F1-score and Precision were also consistent.
S11: onto the first extension: model comparison. Contrary to what one might expect, Random Forest significantly outperformed XGBoost. RF achieved an average AUC of 0.7412, compared to 0.7226 for XGBoost. The difference in F1-score was also notable. Here, a paired t-test to check the statistical significance of this result. The test yielded a t-value of 3.63 and a p-value of 0.0055. Since this p-value is much less than 0.05, we can conclude that the performance.
S12: the second extension used SHAP for feature importance analysis. found that both models agree on the importance of "TYPE_DECLARATION," highlighting that type-level refactorings often require synchronized /ˈsɪŋkrənaɪzd/ test changes. While the models differed in their focus. XGBoost heavily weighted code structure and developer experience, suggesting it captures complex semantic patterns. Random Forest showed a more balanced use of features, including refactoring scale, like the number of files, and other code elements. This gives us insight into their different decision-making processes.
S13: In conclusion, first, the replication confirms that co-occurring refactoring is a stable phenomenon in software development. Second, Random Forest outperforms XGBoost in this specific task, demonstrating that more advanced models do not always guarantee better performance. Third, SHAP analysis identified file history, refactoring scale, and developer experience as key predictive features, providing clear guidance for building intelligent refactoring tools.
S14: and the limitations, limited to Java projects, and the data is historical. There's also potential for more extensive tuning of XGBoost. validate these findings across multiple programming languages, compare more modern models... That's all, thank you for your attention.