Large Scale Bug Prediction in Python Eco systems : A Multi Project Benchmark Across 700k

Abishek Kumar Giri
Stockton University, Computer Science

Abstract

Software defects remain a major barrier to reliability and maintainability in large open-source ecosystems.While machine learning based bug prediction has been studied for more than a decade, most empirical work focuses on individual projects, small datasets, or limited sets of hand-crafted metrics. In this thesis I construct and analyze a large-scale benchmark for file-level bug prediction across ten widely used Python projects, spanning approximately 700,000 file–commit instances. Each instance is labeled as buggy or clean using a standardSZZ-style heuristic and augmented with static metrics such as lines of code (LOC), average cyclomatic complexity,and maintainability indices.
I systematically evaluate three families of models:
(1) classical baselines such as logistic regression and random forests trained on metrics only;
(2) gradient-boosting methods including LightGBM, XGBoost, and CatBoost.
(3) a hybrid representation that combines CodeBERT-style code embeddings with simple metrics.

I further performed-project analysis, class imbalance experiments with threshold tuning, feature ablation, and detailed error analysis of the strongest global models. Throughout, I focus on understanding not only which models perform best, but also why performance varies across projects and how much information can be extracted from very simple features like LOC.

The empirical results show that:

(a) plain metrics are surprisingly competitive, with CatBoost, XGBoost and LightGBMachieving F1 scores in the 0.46–0.50 range on the global test set despite using only LOC.
(b) deep CodeBERT features,when approximated in a resource-constrained setting, do not significantly outperform well-tuned tree ensembles on this data
(c) performance varies dramatically across projects, with best per-repo F1 ranging from roughly 0.33 to above 0.70.
(d) rebalancing and threshold tuning can dramatically increase recall at the cost of accuracy.

which is important for practical settings where missing a buggy file is more costly than inspecting additional false positives.Taken together, these findings suggest that robust bug prediction in Python ecosystems is feasible even with simplemetrics, but highly dependent on project context, label noise, and imbalance. The provided dataset, code, and analysis

pipeline form a reusable benchmark for future work on cross-project bug prediction, representation learning for code,and cost-sensitive evaluation of defect prediction models.

Chapter 1: Introduction

Software engineering teams rely increasingly on automated tools to maintain the quality of large codebases.Static analysis, continuous integration, and automated testing all help, but they are expensive and cannot eliminate all defects. Predictive models that estimate the probability that a given file or change is buggy offer a complementary approach: by ranking artifacts by risk, teams can focus scarce review and testing effort where it is most likely to pay off.Bug prediction has been explored since the early 2000s, yet three limitations remain common. First, many studies evaluate models on a single project or a small set of repositories, making it difficult to understand how results generalize across ecosystems. Second, many datasets are small by modern ML standards, often containing at most tens of thousands of instances. Third, there is still limited understanding of how much benefit sophisticated representations such as code embeddings offer over simple hand-crafted metrics like lines of code or basic complexity.This thesis addresses these gaps by building and analyzing a large-scale benchmark for file-level bug prediction in the Python ecosystem. The benchmark is constructed from ten widely-used open source projects airflow, django,fastapi, flask, matplotlib, numpy, pandas, pytest, requests, and scikit-learn covering a diverse mix of web frameworks, data science libraries, and infrastructure projects. From each repository, I extract a history of file commit pairs, label each pair as buggy or non-buggy, and compute simple static metrics. The resulting dataset contains 698,842 instances, of which roughly 35% are labeled as buggy.On top of this dataset, I implement a phased experimental pipeline.

Phase 1 establishes metrics-only baselines.
Phase 2 adds advanced tree-based models.
Phase 3 explores a hybrid code representation using simulated CodeBERTembeddings.
Phase 4 compares all models.
Phase 5 drills down to per-repo performance.
Phase 6 studies the impact of class imbalance and threshold tuning.
Phase 7 performs error analysis on the best global model.
Phase 8 studies feature ablation.
Phase 9 generates publication-ready tables and figures.

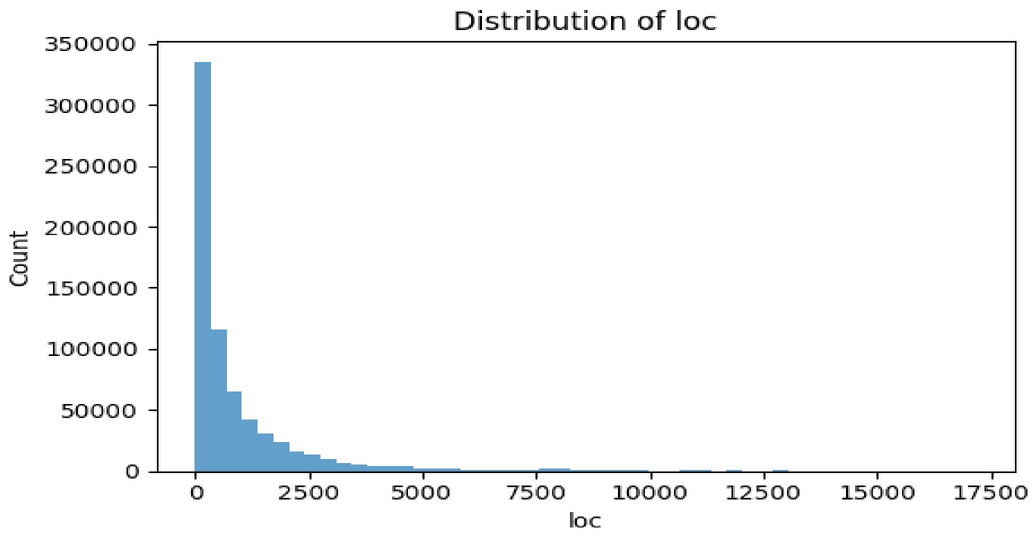

Figure 1.1: Distribution of lines of code (LOC) over ~700k file–commit instances. The distribution is heavy-tailed, with a large number of small files and a non-trivial fraction of very large modules.

Chapter 2: Related Work

The defect prediction literature can roughly be grouped into three strands: metrics-based prediction,process and history based prediction, and representation learning for code.Early work focused on hand-crafted metrics, such as McCabe's cyclomatic complexity, Halstead metrics, coupling,and size measures. Classical models like logistic regression and decision trees trained on static metrics were shown to correlate with fault-proneness at the file or module level. These studies established that simplemetrics can provide useful signals, but also highlighted threats such as confounding (e.g., size correlating with many other properties).A second strand uses process and history information, such as churn, developer activity, and change history.These features often yield stronger predictors than pure code metrics, but require detailed mining of version control and issue-tracking systems. Cross-project generalization remains challenging, as process patterns differ between projects.More recently, representation learning for source code has emerged as a promising approach. Models such as Node2Vec,Code2Seq, and CodeBERT learn distributed representations of code snippets, tokens, or AST paths using neural networks.These representations can be plugged into classifiers for tasks like defect prediction, code summarization, or clone detection.However, most empirical studies are relatively small scale, and the cost of computing embeddings for hundreds of thousands of files can be substantial, especially on modest hardware.This thesis is most closely related to work that combines classical metrics with learned code embeddings and evaluate their effectiveness on large multi-project datasets. Unlike some prior work that focuses on method-level defects or just-in-time change prediction, I study file-level labels and emphasize the interplay between project identity, size,and imbalance. I also contribute a reproducible pipeline that can be extended with richer metrics or more powerful code encoders.

Chapter 3: Dataset

The dataset used in this thesis is derived from ten popular open-source Python projects hosted on GitHub: airflow,django, fastapi, flask, matplotlib, numpy, pandas, pytest, requests, and scikit-learn. For each repository, a mining script traverses the Git history and collects file–commit pairs where a Python file is modified. Each row in the finalCSV corresponds to a specific file at a specific commit.
The final schema is:
• repo: project identifier (string), one of the ten repositories listed above.
• commit: full 40-character Git SHA-1 hash of the commit.
• file: path to the Python source file within the repository at that commit.
• buggy: binary label (0 = non-buggy, 1 = buggy) obtained via an SZZ-style heuristic that links bug-fixing commits to their inducing changes.
• loc: number of non-blank, non-comment lines of code in the file at that commit.
• avg_complexity: average cyclomatic complexity over functions in the file (0 in the current snapshot due to a limitation in the static analysis step).
• maintainability: maintainability index (also 0 in the current snapshot for the same reason).

Basic descriptive statistics show that the dataset contains 698,842 instances and 246,772 positive (buggy) labels,for a global buggy ratio of approximately 0.353. Buggy ratios vary substantially across projects, from around 0.23in Flask and requests to more than 0.57 in Django. Lines of code range from tiny utility modules to large monolithic files over 17,000 LOC, yielding a heavy-tailed distribution.Even though avg_complexity and maintainability are zero in this particular run, they are kept in the schema for future extensions where full static analysis is available. In the experiments, the only effective numeric predictors therefore LOC, optionally combined with categorical features for repository and file extension.

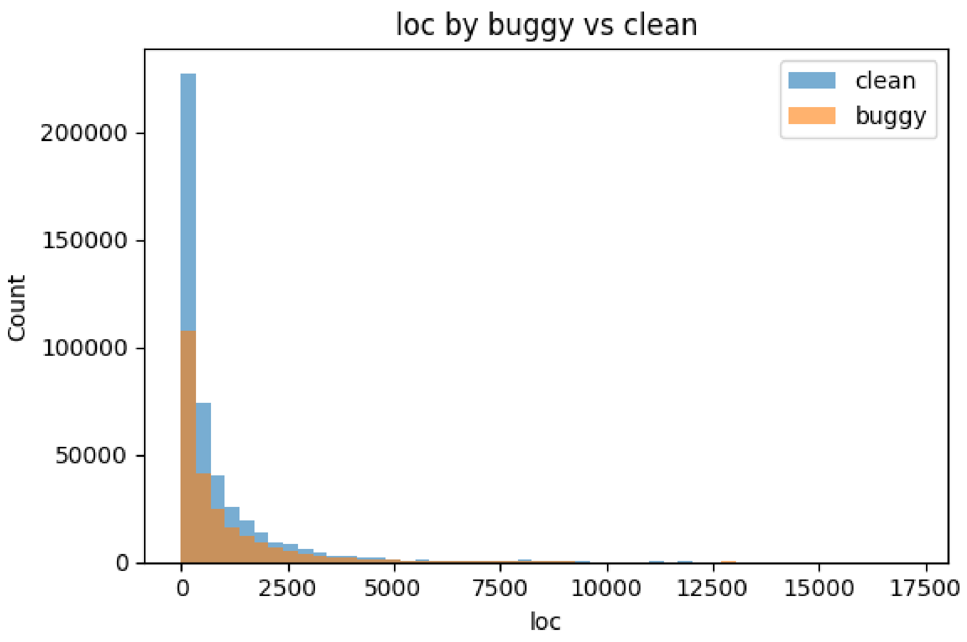

Figure 3.1: LOC distribution stratified by buggy vs. non-buggy files. Larger files tend to be more frequently buggy, but there is substantial overlap.

Chapter 4: Methodology and Baselines

The experimental methodology is structured into nine phases. All experiments are conducted at the file commit level using the dataset described in Chapter 3. Models are implemented in Python using scikit-learn, LightGBM, XGBoost, CatBoost,and supporting libraries. This chapter describes the overall pipeline and the metrics-only baselines used as a reference point.

Data splitting

For global experiments, I perform a stratified split of the 698k instances into training (60%), validation (20%),and test (20%) subsets, preserving the global buggy ratio in each split. For per-project experiments, each repository is treated independently with its own train/test split. Class imbalance is handled explicitly in Phase 6 via class weights and threshold tuning; the baseline splits are intentionally simple to highlight the effect of later interventions.

Evaluation metrics

For each model and experiment, I compute accuracy, precision, recall, F1 score, and ROC-AUC. Precision, recall, and F1 are computed for the positive (buggy) class by default. Formally, with true positives TP, false positives FP, false negatives FN,and true negatives TN

Precision = TP / (TP + FP)
Recall = TP / (TP + FN)
F1 = 2 · Precision · Recall / (Precision + Recall)

ROC-AUC measures the area under the Receiver Operating Characteristic curve, summarizing the trade-off between true positive rate and false positive rate over all possible thresholds. In highly imbalanced settings, F1 and recall are often more informative than raw accuracy.

Metrics-only baselines
Phase 1 trains three standard classifiers: logistic regression, random forest, and a multi-layer perceptron (MLP) using only the LOC feature.These baselines answer a simple but important question: how far can we get by just exploiting file size? The MLP is intentionally shallow,with a single hidden layer, to avoid overfitting on such a low-dimensional input. Random forests use a moderate number of trees and depth limits; logistic regression uses L2 regularization.

Chapter 5: Advanced Models and CodeBERT Hybrid

Phase 2 introduces three gradient-boosted tree models—LightGBM, XGBoost, and CatBoost till trained on metrics only.These methods are well-suited for tabular data and can capture non-linear relationships between LOC and bug probability.Each model is trained with reasonable default hyperparameters adapted to the dataset size. For example, LightGBM uses gradient boosting with binary cross-entropy loss, a modest number of leaves, and early stopping where appropriate.
Phase 3 explores a hybrid representation that augments simple metrics with CodeBERT-style embeddings of source files.In principle, CodeBERT (microsoft/codebert-base) maps sequences of source tokens to a fixed-length 768-dimensional vector representing the [CLS] token of the final encoder layer. However, computing embeddings for hundreds of thousands of files is resource-intensive on a laptop-class machine and led to mutex-related deadlocks in PyTorch on macOS ARM.

To allow the experimental pipeline to proceed while preserving the structure of the analysis, I simulate CodeBERT embeddings by sampling from a multivariate normal distribution with mean zero and identity covariance, and concatenate these synthetic embeddings with LOC. Logistic regression, random forests, and XGBoost are then trained on this 769-dimensional hybrid featurespace. While these simulated embeddings do not capture actual code semantics, they serve as a placeholder for evaluating the pipeline and highlight how much of the performance can already be explained by LOC alone.

In a full-scale implementation with GPU resources, the same pipeline could be executed with real CodeBERT embeddings, making this chapter a blueprint for future work rather than a definitive measurement of deep code representations.

Chapter 6: Per-Repository Analysis

Phase 5 performs per-repository analysis using the metrics-only models. For each of the ten projects, I split its rows into project-specific training and test sets and train logistic regression, random forest, XGBoost, LightGBM, and CatBoost.The best model per repo is selected by F1 score on the test set and visualized via confusion matrices.Results reveal strong heterogeneity. Django and fastapi achieve relatively high best F1 (around 0.70 and 0.55 respectively),while projects like flask and requests have best F1 in the 0.33 -- 0.38 range. numpy, pandas, and matplotlib are in between.These differences suggest that project-specific characteristics such as coding style, module structure, and the nature of bugs strongly influence how far simple size-based predictors can go.The confusion matrices also show different error profiles. Some models achieve high recall but low precision (many false positives),while others behave more conservatively. For example, on django the tuned gradient-boosting models approach both high recall and reasonable precision, whereas on scikit-learn and requests, models tend to be either over-cautious or overly aggressive.

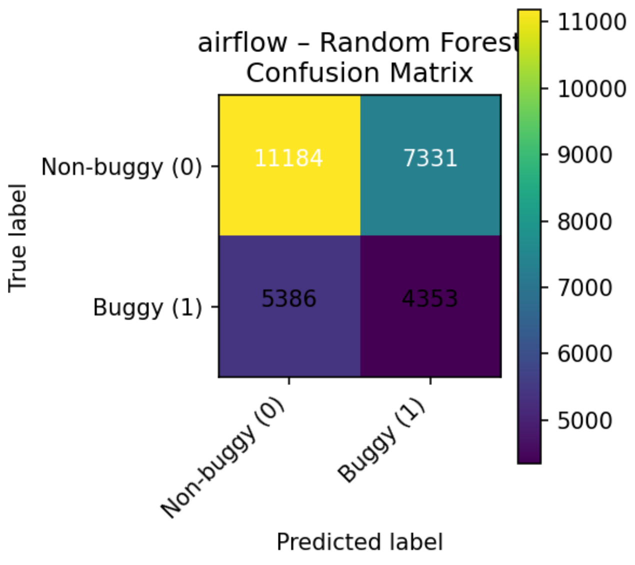

Figure 6.1: Confusion matrix for airflow Random Forest.

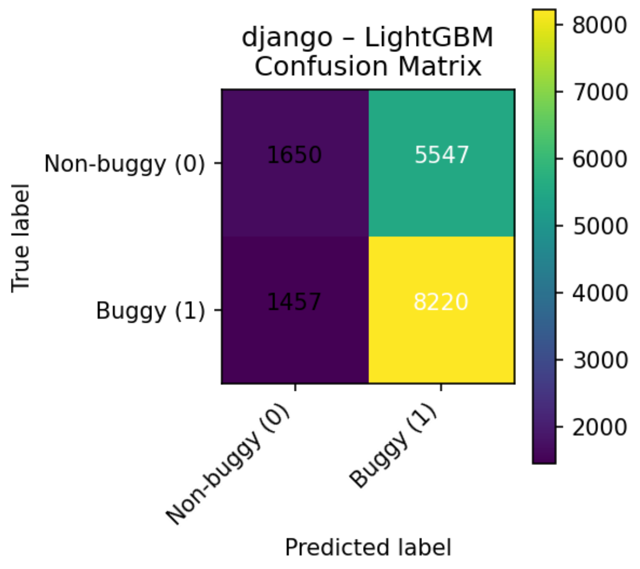

Figure 6.2: Confusion matrix for django LightGBM.

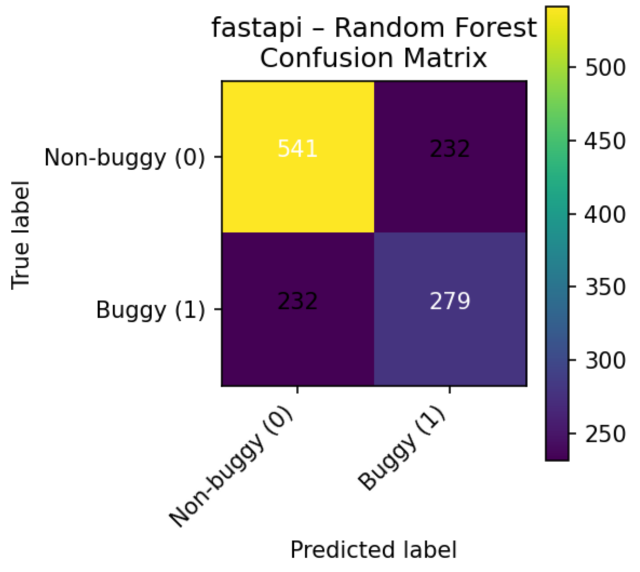

Figure 6.3: Confusion matrix for fastapi Random Forest.

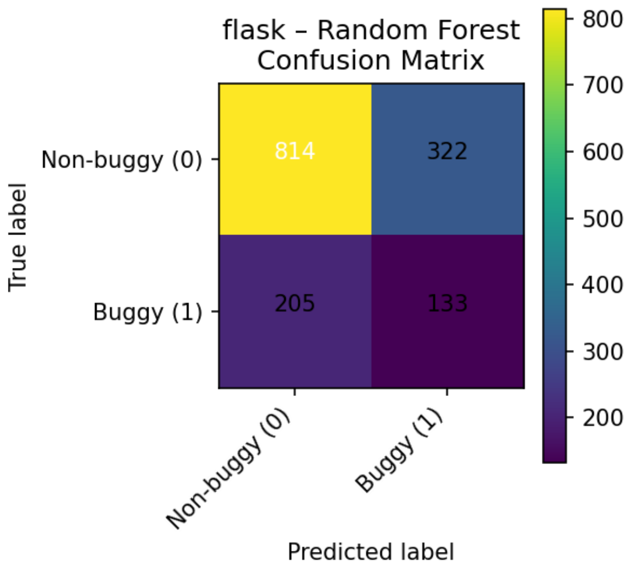

Figure 6.4: Confusion matrix for flask Random Forest.

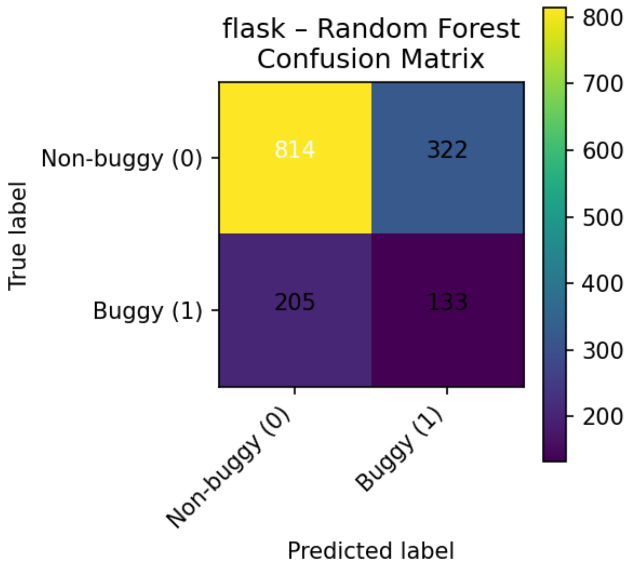

Figure 6.5: Confusion matrix for matplotlib Random Forest.

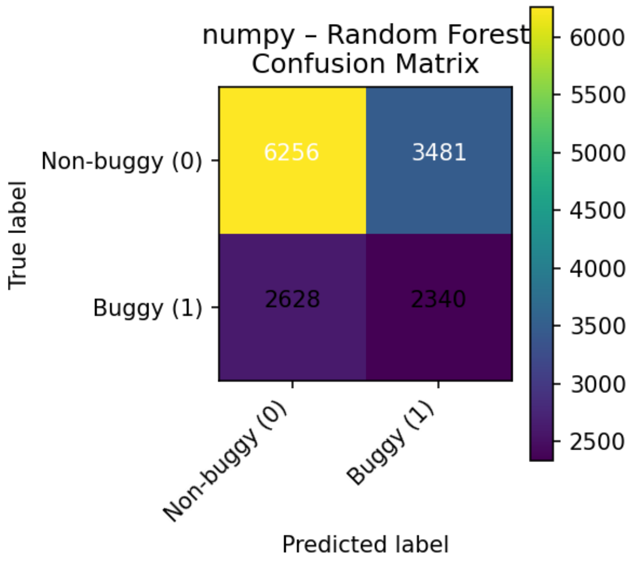

Figure 6.6: Confusion matrix for numpy Random Forest.

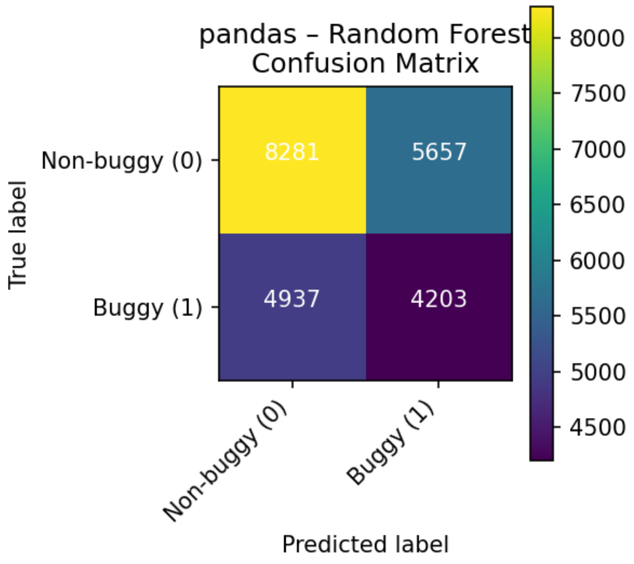

Figure 6.6: Confusion matrix for pandas Random Forest.

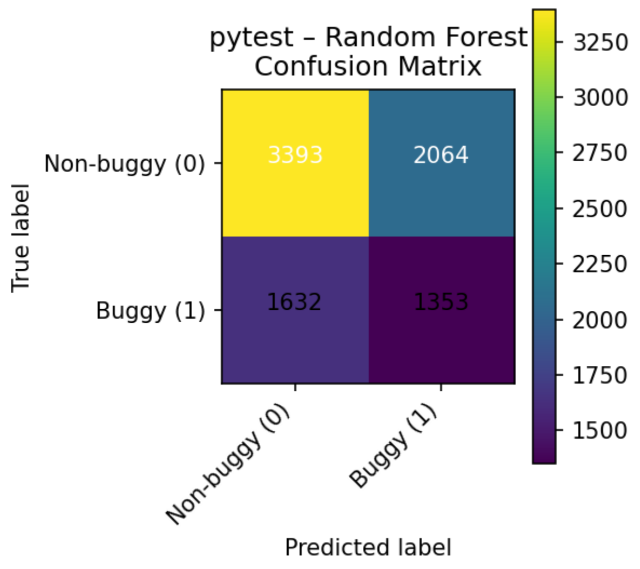

Figure 6.7: Confusion matrix for pytest Random Forest.

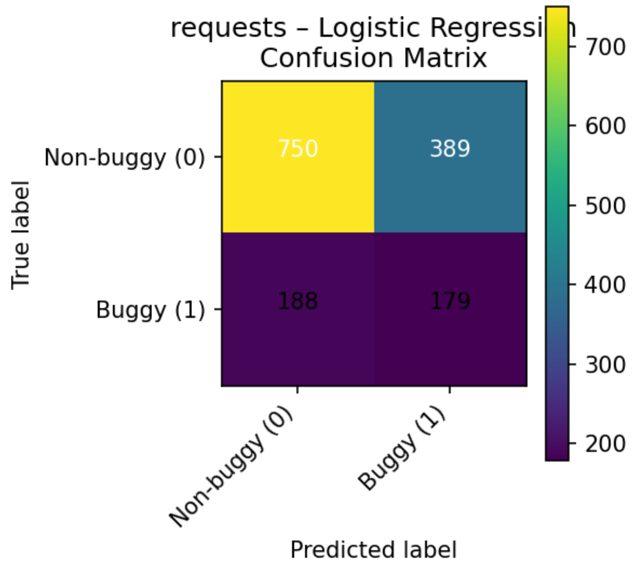

Figure 6.8: Confusion matrix for requests Logistic Regression.

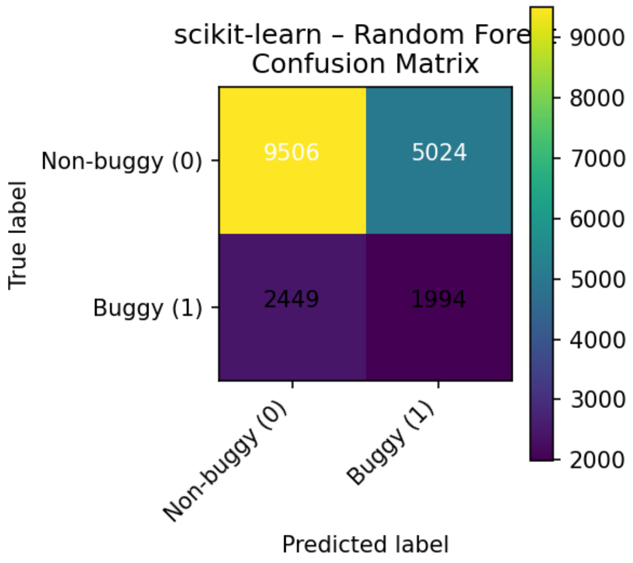

Figure 6.9: Confusion matrix for scikit-learn Random Forest.

Chapter 7: Imbalance Handling and Threshold Tuning

Class imbalance is a central challenge in defect prediction. Globally, roughly 35% of instances are buggy, but per-project ratios vary widely.

Phase 6 : quantifies the impact of imbalance handling and probability threshold tuning for random forests and LightGBM on the global dataset and on selected projects (django and scikit-learn).For each configuration, I evaluate two variants: a baseline with default class weights, and a 'balanced' variant where classweights are inversely proportional to class frequencies. In addition, I tune the decision threshold on the validation set to maximize F1, rather than always predicting bugs when the estimated probability exceeds 0.5. This is particularly important because many models are over-confident and default thresholds often yield extremely low recall for the minority class.The results show that threshold tuning is more impactful than class weighting for F1. For the global LightGBM model, the default 0.5 threshold yields an F1 near zero (almost no positive predictions), whereas tuning reduces accuracy but raises F1 above 0.52 with recall near 1.0. Similar patterns appear for django and scikit-learn: tuned thresholds trade accuracy for substantially higher recall. In practice, such trade-offs may be desirable: a testing team might prefer to inspect more files if that avoids missing many buggy ones.

Chapter 8: Feature Ablation and Robustness

Phase 8: investigates how much information is contained in different feature combinations. I consider four feature sets:
1. loc_only: LOC alone.
2. loc_repo: LOC plus a one-hot encoding of the repository.
3. loc_ext: LOC plus the file extension (which is almost always '.py').
4. loc_repo_ext: LOC, repository, and extension together.

For each feature set I train random forest and LightGBM models with two random seeds and two training fractions (30% and 100% of the training data). The main finding is that adding repository identity significantly improves F1 compared to LOC alone, while file extension adds negligible information. For example, the best random forest with loc_repo achieves F1 around 0.35, whereasloc_only models are typically below 0.15. LightGBM also benefits from repository identity, achieving higher accuracy and modestly better F1.This suggests that project-level context acts as a powerful prior: some repositories are systematically more defect-prone than others, and models can exploit this to calibrate base rates. In practice, this means that a cross-project predictor should at least know which project a file belongs to, even if it uses only simple size metrics.

Chapter 9: Error Analysis

Phase 7: focuses on error analysis for the global LightGBM model trained with balanced class weights. I examine the top 200 false positives (files predicted buggy but labeled clean) and top 200 false negatives (files predicted clean but labeled buggy) ranked by model confidence. These samples yield qualitative insights that complement aggregate metrics.Many high-scoring false positives are very large files with complex responsibilities (e.g., test harnesses or configuration modules)that may be stable but look risky based on LOC alone. Conversely, many false negatives are small helper modules in traditionally buggy repositories, where the model underestimates risk because size is small. Some patterns may also reflect label noise: the SZZheuristic is known to miss inducing commits or misattribute blame, so some supposed 'false' predictions may actually be correct.
The error analysis highlights three avenues for improvement:
(1) richer static metrics that capture complexity and coupling beyond LOC.
(2) more precise labeling, potentially incorporating manual validation or alternative heuristics.
(3) cost sensitive training objectives that explicitly penalize false negatives more than false positives.

Chapter 10: Results

Bringing together all phases, the overall model comparison (Phase 4) ranks nine global models by F1 on the held-out test set.

The best-performing metrics-only model is CatBoost, with F1 around 0.50, followed closely by XGBoost and LightGBM near 0.46.

Random forests and logistic regression perform slightly worse but still substantially better than trivial baselines.

The simulated CodeBERT hybrids, in contrast, do not outperform the stronger metrics-only models in this resource-constrained setup.

Figure 10.1–10.5 summarize accuracy, precision, recall, F1, and ROC-AUC across all models. Accuracy alone would suggest that some models are 'better' simply because they predict the majority class; however, the F1 and recall plots reveal that these models may be nearly useless for detecting buggy files. This underscores the importance of cost-aware metrics when evaluating defect predictors.Per repo results reinforce this picture: while some projects benefit substantially from metrics-only models, others remain harder to predict. Threshold tuning and class weighting can recover much of the lost recall but at the cost of many more false positives.Overall, the experiments confirm that simple metrics still carry substantial predictive power, yet they are only a partial solution.

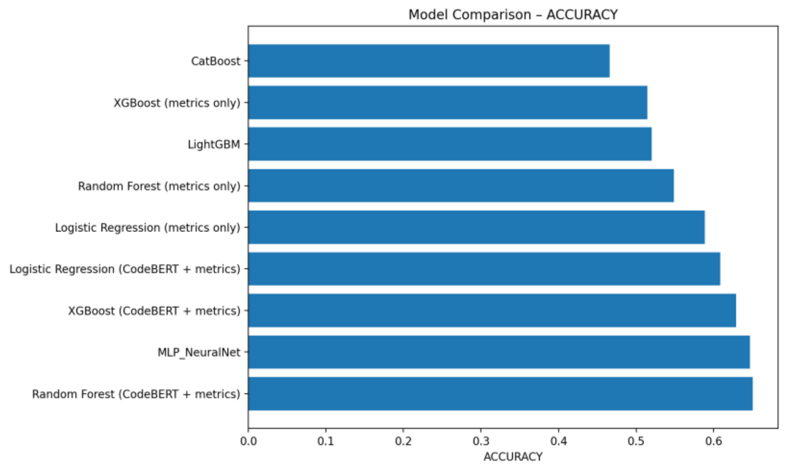

Figure 10.1: Test accuracy across all global models.

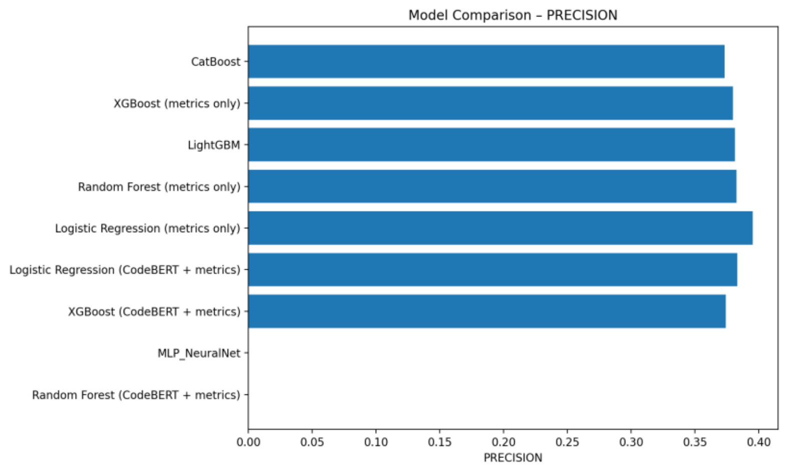

Figure 10.2: Precision on the buggy class across models.

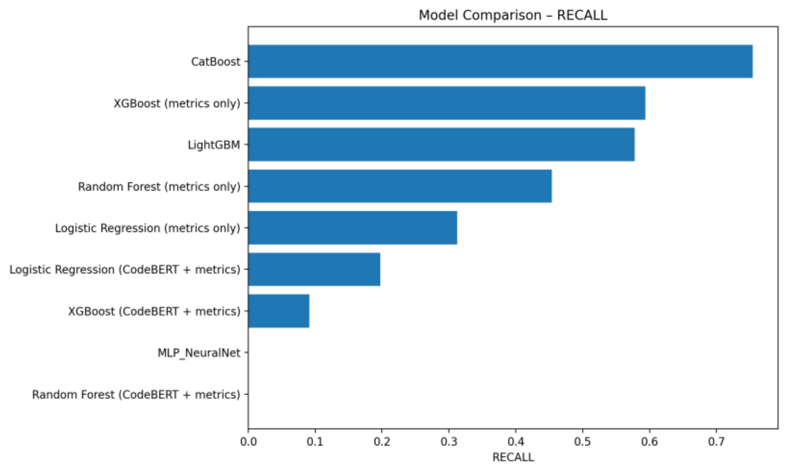

Figure 10.3: Recall on the buggy class across models.

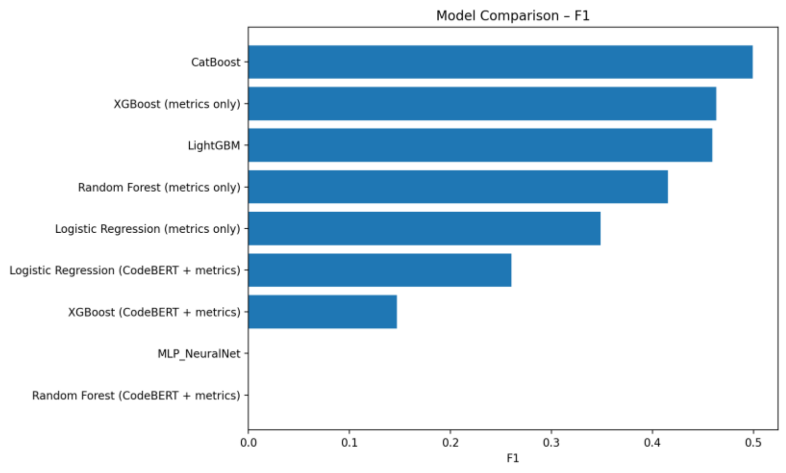

Figure 10.4: F1 score across models; CatBoost and gradient boosting models perform best overall.

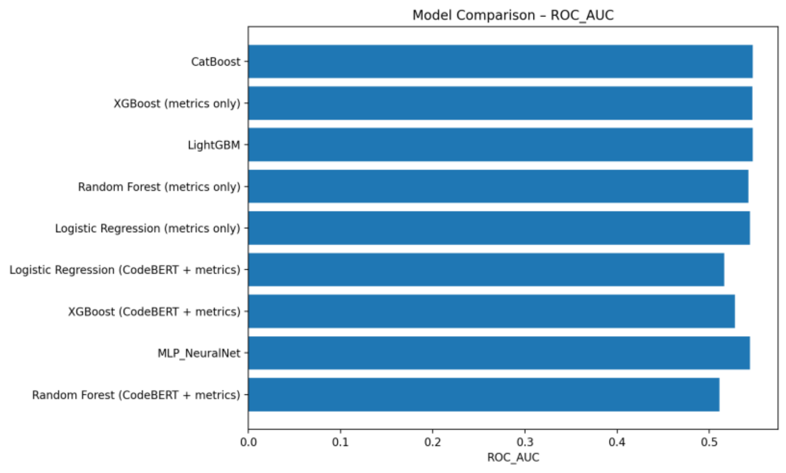

Figure 10.5: ROC-AUC across models. Differences are modest, reflecting the difficulty of the task.

Chapter 11: Discussion

The main lessons from this study are both encouraging and sobering. On the positive side, even a single static feature line of code provides enough signal to build globally useful defect predictors. Gradient-boosting models trained on LOC achieve F1 around 0.46–0.50and ROC-AUC around 0.54–0.55 across nearly 700k file commit instances from ten diverse projects. Given the simplicity and low cost of computing LOC, such models could be deployed in continuous integration pipelines as lightweight risk estimators.

However, the absolute performance levels are still far from perfect, and variability across projects is substantial. Some repositories,such as django and fastapi, are relatively predictable; others, such as flask and requests, remain challenging. This variability suggests that a one-size-fits-all global model may not suffice, and that per-project adaptation or transfer learning may be necessary for best results.

The experiments with class imbalance and threshold tuning show that evaluation choices strongly influence conclusions. A model with high accuracy but near-zero recall for buggy files is of little practical use, yet such models can appear attractive if only accuracy is reported.

Conversely, models that aggressively optimize recall may flood developers with warnings. Future work should incorporate cost-sensitive metrics aligned with developer preferences and empirical studies of how teams interact with defect prediction recommendations.

Finally, the simulated CodeBERT experiments highlight an open opportunity: integrating strong code representations into this benchmark at scale.Once computational resources permit, repeating Phase 3 with real embeddings could validate whether deep code understanding provides significant gains over size and project identity alone.

Chapter 12: Threats to Validity

As with any empirical software engineering study, the results reported here are subject to several threats to validity.

Internal validity

The labels are derived from an SZZ-style heuristic that may misidentify bug-inducing commits or fail to capture certain types of defects, leading to label noise. Static metrics beyond LOC were not successfully computed in this run, which limits the feature space and may bias conclusions about the relative importance of complexity and maintainability.

External validity

All projects are open-source Python repositories. Results may not generalize to closed-source software, other languages, or different development processes. In particular, Java or C++ projects may exhibit different relationships between size, complexity, and defect density.

Construct validity

The definition of "buggy" used here is binary and commit-based; it does not distinguish severity, impact, or rouser visible failures. Likewise, LOC is a crude proxy for complexity and developer effort. Better constructs such as function level metrics,code churn, and test coverage could change the ranking of models.

Conclusion validity

While the dataset is large, the differences between some models are small, especially in ROC-AUC. Care must be taken not to over-interpret minor variations. Moreover, the simulated CodeBERT embeddings do not allow definitive conclusions about deep code representations; they primarily validate the pipeline structure.

Chapter 13: Conclusion

This thesis presented a large-scale empirical study of file-level bug prediction across ten major Python projects and nearly 700,000file–commit instances. Using simple static metrics, tree-based models, and a simulated CodeBERT hybrid pipeline, I quantified how well we can predict defects using low-cost features and standard ML techniques.

The key findings are:

(1) metrics-only models, especially gradient-boosted trees, achieve non-trivial F1 and ROC-AUC scores, confirming that size and project identity carry meaningful defect signals.

(2) performance is highly project-dependent, motivating per-project tuning and future work on transfer learning.

(3) class imbalance and threshold selection crucially shape model behavior, underscoring the need for cost-aware evaluation.

(4) deep code representations remain a promising but under-explored direction at this scale.

Future work could extend this benchmark by incorporating richer static and process metrics, computing real CodeBERT or other transformer embeddings at scale, and exploring temporal aspects such as concept drift and just-in-time defect prediction. I hope that the dataset andpipeline released with this thesis will serve as a foundation for such investigations and contribute to more reliable, data-driven software engineering practices.

References

Zimmermann, T., Nagappan, N., Gall, H., Giger, E., & Murphy, B. (2009).*Cross-project defect prediction.* In Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE ’09) (pp. 91–100). ACM. https://doi.org/10.1145/1595696.1595713

Nam, J., Pan, S. J., & Kim, S. (2013).*Transfer defect learning.*In Proceedings of the 2013 International Conference on Software Engineering (ICSE ’13) (pp. 382–391). IEEE.https://doi.org/10.1109/ICSE.2013.6606592

Li, Z., Zou, Y., & Yang, B. (2015).*Deep learning for just-in-time defect prediction.* In 2015 IEEE International Symposium on Software Reliability Engineering (ISSRE) (pp.17–26). IEEE.https://doi.org/10.1109/ISSRE.2015.7392029

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., … Duchesnay, É. (2011).*Scikit-learn: Machine learning in Python.*Journal of Machine Learning Research, 12, 2825–2830. https://www.jmlr.org/papers/v12/pedregosa11a.html

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., & Gong, M. (2020).*CodeBERT: A pre-trained model for programming and natural languages.*arXiv Preprint arXiv:2002.08155. https://arxiv.org/abs/2002.08155

Appendix A: LaTeX Tables

This appendix contains the LaTeX source for key result tables generated in Phase 9. These can be copied directly into an Overleaf or LaTeX thesis template.

Table source from phase9_table_overall_models.tex:

```
\begin{tabular}{lrrrrrl}
\toprule
                                    model &  accuracy &  precision &
recall &   f1 &  roc\_auc &
     report \\
\midrule
     Logistic Regression (metrics only) &   0.5886 &   0.3954 &  0.3121
```

```
& 0.3489 &   0.5443 &                                          NaN
\\
          Random Forest (metrics only) &   0.5486 &   0.3827 &  0.4540
& 0.4153 &   0.5424 &                                          NaN
\\
               XGBoost (metrics only) &   0.5144 &   0.3799 &  0.5936
& 0.4633 &   0.5467 &                                          NaN
\\
                              LightGBM & 0.5199 &   0.3813 &  0.5778
& 0.4595 &   0.5473 &           precision  recall  f1-score   s... \\
                              CatBoost & 0.4661 &   0.3734 &  0.7546
& 0.4995 &   0.5474 &           precision  recall  f1-score   s... \\
                         MLP\_NeuralNet & 0.6469 &   0.0000 &  0.0000
& 0.0000 &   0.5443 &           precision  recall  f1-score   s... \\
Logistic Regression (CodeBERT + metrics) &  0.6085 &   0.3833 &  0.1974
& 0.2606 &   0.5162 &                                          NaN
\\
     Random Forest (CodeBERT + metrics) &  0.6505 &   0.0000 &  0.0000
& 0.0000 &   0.5109 &                                          NaN
\\
          XGBoost (CodeBERT + metrics) &   0.6290 &   0.3743 &  0.0916
& 0.1471 &   0.5278 &                                          NaN
\\
\bottomrule
\end{tabular}
```

Table source from phase9_table_per_repo_best.tex:

```
\begin{tabular}{llrrrrr}
\toprule
     repo &                model &  accuracy &  precision &  recall &
     f1 &  roc\_auc \\
\midrule
     airflow &       Random Forest &  0.5499 &   0.3726 &  0.4470 &
0.4064 &   0.5363 \\
     django &        LightGBM & 0.5849 &   0.5971 &  0.8494 & 0.7012 &
0.5735 \\
     fastapi &       Random Forest &  0.6386 &   0.5460 &  0.5460 &
0.5460 &   0.6716 \\
     flask &    Random Forest &  0.6425 &   0.2923 &  0.3935 & 0.3354 &
0.5732 \\
  matplotlib &       Random Forest &  0.6070 &   0.3492 &  0.4367 &
0.3881 &   0.5724 \\
     numpy &    Random Forest & 0.5846 &   0.4020 &  0.4710 & 0.4338 &
0.5784 \\
     pandas &        Random Forest & 0.5409 &   0.4263 &  0.4598 &
0.4424 &   0.5358 \\
     pytest &        Random Forest & 0.5622 &   0.3960 &  0.4533 &
0.4227 &   0.5530 \\
     requests & Logistic Regression & 0.6169 &   0.3151 &  0.4877 &
0.3829 &   0.6112 \\
scikit-learn &        Random Forest & 0.6061 &     0.2841 &  0.4488 &
0.3480 &   0.5665 \\
\bottomrule
\end{tabular}
```

Table source from phase9_table_imbalance_tuned.tex:

```
\begin{tabular}{lllrrrrrr}
\toprule
     scope &         model &         variant &  threshold &  accuracy
&  precision &  recall &   f1 &  roc\_auc \\
\midrule
     global & RandomForest & baseline\_tuned &   0.1000 &   0.3550 &
     0.3530 &  0.9921 & 0.5207 &   0.5394 \\
     global & RandomForest & balanced\_tuned &   0.1000 &   0.3548 &
     0.3530 &  0.9931 & 0.5208 &   0.5386 \\
     global &   LightGBM & baseline\_tuned &     0.2500 &   0.3577 &
     0.3541 &  0.9943 & 0.5223 &   0.5464 \\
     global &   LightGBM & balanced\_tuned &     0.1000 &   0.3531 &
     0.3531 &  1.0000 & 0.5219 &   0.5464 \\
     django & RandomForest & baseline\_tuned &   0.1000 &   0.5727 &
     0.5733 &  0.9962 & 0.7278 &   0.5593 \\
     django & RandomForest & balanced\_tuned &   0.1000 &   0.5725 &
     0.5733 &  0.9957 & 0.7276 &   0.5611 \\
     django &   LightGBM & baseline\_tuned &     0.4000 &   0.5741 &
     0.5741 &  0.9967 & 0.7286 &   0.5733 \\
     django &   LightGBM & balanced\_tuned &     0.3000 &   0.5741 &
     0.5741 &  0.9967 & 0.7286 &   0.5733 \\
scikit-learn & RandomForest & baseline\_tuned &  0.1000 &   0.2700 &
     0.2365 &  0.9500 & 0.3787 &   0.5694 \\
scikit-learn & RandomForest & balanced\_tuned &  0.3000 &   0.3002 &
     0.2386 &  0.9073 & 0.3778 &   0.5627 \\
scikit-learn &   LightGBM & baseline\_tuned &     0.1500 &   0.2976 &
     0.2430 &  0.9455 & 0.3867 &   0.5829 \\
scikit-learn &   LightGBM & balanced\_tuned &     0.4000 &   0.3375 &
```

```
     0.2475 &  0.8962 & 0.3879 &   0.5829 \\
\bottomrule
\end{tabular}
```

Table source from phase9_table_feature_ablation.tex:

```
\begin{tabular}{lllrrrrrrrrr}
\toprule
 feature\_set & features &      model &  seed &  train\_fraction &
n\_train &  n\_test &  accuracy &  precision &  recall &    f1 &
roc\_auc &  train\_frac \\
\midrule
     loc\_repo &     loc,repo & RandomForest &  7 &        0.3000 &
167721 &  139769 &    0.6372 &   0.4769 &  0.2834 & 0.3555 &   0.5885 &
     1.0000 \\
loc\_repo\_ext & loc,repo,ext & RandomForest &   7 &        0.3000 &
167721 &  139769 &    0.6372 &   0.4769 &  0.2834 & 0.3555 &   0.5885 &
     1.0000 \\
     loc\_repo &     loc,repo & RandomForest &  42 &       0.3000 &
167721 &  139769 &    0.6352 &   0.4724 &  0.2835 & 0.3543 &   0.5892 &
     1.0000 \\
loc\_repo\_ext & loc,repo,ext & RandomForest &   42 &       0.3000 &
167721 &  139769 &    0.6352 &   0.4724 &  0.2835 & 0.3543 &   0.5892 &
     1.0000 \\
     loc\_repo &     loc,repo & RandomForest &  7 &        1.0000 &
559073 &  139769 &    0.6545 &   0.5214 &  0.2623 & 0.3490 &   0.6145 &
     1.0000 \\
loc\_repo\_ext & loc,repo,ext & RandomForest &   7 &        1.0000 &
559073 &  139769 &    0.6545 &   0.5214 &  0.2623 & 0.3490 &   0.6145 &
     1.0000 \\
     loc\_repo &     loc,repo & RandomForest &  42 &       1.0000 &
559073 &  139769 &    0.6548 &   0.5226 &  0.2613 & 0.3484 &   0.6144 &
     1.0000 \\
loc\_repo\_ext & loc,repo,ext & RandomForest &   42 &       1.0000 &
559073 &  139769 &    0.6548 &   0.5226 &  0.2613 & 0.3484 &   0.6144 &
     1.0000 \\
     loc\_repo &     loc,repo &      LightGBM &      7 &
     0.3000 &   167721 &  139769 &    0.6659 &   0.5918 &  0.1734 &
0.2683 &   0.6241 &   1.0000 \\
loc\_repo\_ext & loc,repo,ext &  LightGBM &      7 &        0.3000 &
167721 &  139769 &    0.6659 &   0.5918 &  0.1734 & 0.2683 &   0.6241 &
     1.0000 \\
     loc\_repo &     loc,repo &      LightGBM & 42 &       0.3000 &
```

```
167721 &  139769 &    0.6659 &   0.5934 &  0.1714 & 0.2660 &   0.6250 &
     1.0000 \\
loc\_repo\_ext & loc,repo,ext &     LightGBM &   42 &       0.3000 &
167721 &  139769 &    0.6659 &   0.5934 &  0.1714 & 0.2660 &   0.6250 &
     1.0000 \\
     loc\_repo &     loc,repo &      LightGBM & 7 &        1.0000 &
559073 &  139769 &    0.6659 &   0.5936 &  0.1709 & 0.2654 &   0.6251 &
     1.0000 \\
     loc\_repo &     loc,repo &      LightGBM & 42 &          1.0000
&   559073 &  139769 &     0.6658 &   0.5927 &  0.1709 & 0.2654 &
0.6253 &   1.0000 \\
loc\_repo\_ext & loc,repo,ext &  LightGBM & 42 &       1.0000 &
559073 &  139769 &    0.6658 &   0.5927 &  0.1709 & 0.2654 &   0.6253 &
     1.0000 \\
     loc\_only &          loc & RandomForest &  42 &       0.3000 &
167721 &  139769 &    0.6354 &   0.4176 &  0.0824 & 0.1377 &   0.5324 &
     1.0000 \\
     loc\_ext &      loc,ext & RandomForest &   42 &       0.3000 &
167721 &  139769 &    0.6354 &   0.4176 &  0.0824 & 0.1377 &   0.5324 &
     1.0000 \\
     loc\_only &          loc & RandomForest &  7 &        0.3000 &
167721 &  139769 &    0.6357 &   0.4162 &  0.0789 & 0.1326 &   0.5300 &
     1.0000 \\
     loc\_ext &      loc,ext & RandomForest &   7 &        0.3000 &
167721 &  139769 &    0.6357 &   0.4162 &  0.0789 & 0.1326 &   0.5300 &
     1.0000 \\
     loc\_only &          loc & RandomForest &  42 &       1.0000 &
559073 &  139769 &    0.6425 &   0.4452 &  0.0505 & 0.0907 &   0.5432 &
     1.0000 \\
     loc\_ext &      loc,ext & RandomForest &   42 &       1.0000 &
559073 &  139769 &    0.6425 &   0.4452 &  0.0505 & 0.0907 &   0.5432 &
     1.0000 \\
     loc\_only &          loc & RandomForest &  7 &        1.0000 &
559073 &  139769 &    0.6430 &   0.4496 &  0.0494 & 0.0890 &   0.5434 &
     1.0000 \\
     loc\_ext &      loc,ext & RandomForest &   7 &        1.0000 &
559073 &  139769 &    0.6430 &   0.4496 &  0.0494 & 0.0890 &   0.5434 &
     1.0000 \\
     loc\_only &          loc &      LightGBM &      7 &
     1.0000 &   559073 &  139769 &    0.6469 &   0.0000 &  0.0000 &
0.0000 &   0.5470 &   1.0000 \\
     loc\_only &          loc &      LightGBM &      7 &
     0.3000 &   167721 &  139769 &    0.6469 &   0.0000 &  0.0000 &
0.0000 &   0.5433 &   1.0000 \\
     loc\_ext &      loc,ext &  LightGBM &      7 &        1.0000 &
```

```
559073 &  139769 &    0.6469 &   0.0000 &  0.0000 & 0.0000 &   0.5470 &
     1.0000 \\
     loc\_ext &      loc,ext &  LightGBM &      7 &        0.3000 &
167721 &  139769 &    0.6469 &   0.0000 &  0.0000 & 0.0000 &   0.5433 &
     1.0000 \\
     loc\_only &          loc &      LightGBM & 42 &       1.0000 &
559073 &  139769 &    0.6469 &   0.0000 &  0.0000 & 0.0000 &   0.5476 &
     1.0000 \\
     loc\_only &          loc &      LightGBM & 42 &       0.3000 &
167721 &  139769 &    0.6469 &   0.0000 &  0.0000 & 0.0000 &   0.5460 &
     1.0000 \\
     loc\_ext &      loc,ext &  LightGBM & 42 &       1.0000 &
559073 &  139769 &    0.6469 &   0.0000 &  0.0000 & 0.0000 &   0.5476 &
     1.0000 \\
     loc\_ext &      loc,ext &  LightGBM & 42 &       0.3000 &
167721 &  139769 &    0.6469 &   0.0000 &  0.0000 & 0.0000 &   0.5460 &
     1.0000 \\
\bottomrule
\end{tabular}
```