Proceedings of Machine Learning Research **318** (2026)
The 39th Canadian Conference on Artificial Intelligence

Just-in-Time Defect Prediction Using Cost-Efficient Boosting Models

Elza Jung[†,*], and Md Asif Khan[†,*]
[†] Wilfrid Laurier University, Waterloo, ON, N2L 3C5, Canada

Abstract

Just-in-time (JIT) software defect prediction (SDP) handles imbalanced commit data, reflecting real-world software where bugs are rare. Higher recall and a model's ability to predict defects are crucial in such settings. Recently, many JIT-SDP approaches have been proposed, predominantly utilizing deep-learning (DL) models. However, tuned XGBoost among traditional classifiers, known for cost-efficiency, has not been explored. Therefore, we explore how hyperparameter(HP) tuned and SMOTE-rebalanced XGBoost perform in imbalanced datasets, focusing on AUC-ROC and Recall. Our findings indicate that selecting five key features can be as effective as using fourteen features. We further explain how HP tuning and the over-sampling method improve XGBoost by 1.19%-6.48% in AUC-ROC and 19.32%-43.70% in Recall. Statistical analysis shows that the final XGBoost model achieves the best average performance among the evaluated baselines, with 0.7442 AUC-ROC, 0.4747 F1-Score, and 0.7099 Recall.

Keywords: Just-in-Time Defect Prediction, Boosting Algorithms, Expert Features, Cost-Efficient, Machine Learning

1. Introduction

As software development involves handling multiple program requirements across various languages, Just-in-time Software Defect Prediction (JIT-SDP) became essential to maintain the code quality [1]. However, JIT-SDP is challenged to handle the imbalance between the number of clean commits and the number of baggy commits [2, 3]. Therefore, it is essential for JIT-SDP to have a high precision as well as a high recall. In the recent studies [4–7], JIT-SDP has two parts **1) semantic feature analysis:** This part captures the meaning and structure of code changes by learning representations from source code itself and **2) expert feature analysis:** This part relies on manually drafted metrics, such as change size, history and developer experience. We focused on comparing models based on expert feature analysis.

Recent advancement, JITSmart [4] has demonstrated that combining expert and semantic feature analysis with a Multi-Layer Perceptron (MLP) improved the Recall compared to Random Forest (RF) from CC2Vec [7], Logistic Regression (LR) from LAPredict [8], CodeBERT from JITFine [5] by 19.18%, 30.97%, and 9.44%, respectively. It also improved the Area Under the Receiver Operating Characteristic Curve (AUC-ROC) by 85.77%, 702.74%, and 19.89%, respectively. However, Deep Learning (DL) and MLP have operational costs and cost-inefficient challenges [4, 6]. Several researchers [8, 9] used cost-efficient LR and RF for JIT-SDP. However, Boosting Algorithms were only considered and not used [10]. XGBoost can outperform DL models, especially with tabular data [11]. Therefore, we explore a traditional classifier, XGBoost, focusing on AUC-ROC and Recall.

This paper addresses these gaps by proposing a robust and cost-efficient framework for JIT-SDP using XGBoost.

- **Feature Optimization.** We engineered 5 features and metrics extracted from code changes to train models predicting bug-introducing commits, to provide missing data

[*]jung1603@mylaurier.ca, asikhan@wlu.ca

like tendency of entropy, total volume of changes and balance between additions and deletions.

- **Boosting for Expert Features.** We performed Bayesian hyperparameter (HP) optimization over seven key XGBoost parameters (`max_depth`, `min_child_weight`, `subsample`, `colsample_bytree`, `gamma`, `num_parallel_tree`, and `learning rate`), executing 10 trials per setting. Across 4 different features and sampling configurations, this yielded a total of 40 model training iterations.
- **Cost-Efficiency and Recall Improvement.** We show that XGBoost is not only more cost-efficient than DL models, but also effectively overcomes the low Recall of LR, particularly for defective commits, improving Recall in imbalanced datasets.

The rest of the paper is organized as follows. Section 2 provides an overview of related work on existing JIT-SDP expert features, models, and methods for handling imbalanced datasets. Section 3 introduces our research questions and describes our experimental settings to train the XGBoost. Section 4 reports our experimental results to answer the research questions. Section 5 discusses our findings. Section 6 presents threats to validity. Section 7 concludes the paper with future work.

2. Related Work

2.1. JIT-SDP Expert Features

Many JIT-SDP projects [4–6, 8, 12, 13] used Kamei's 14 expert features, described in Table 1. Kamei's expert features are change-level metrics that describe how large, scattered, and risky a commit is (e.g., lines of code added/deleted/modified, number of subsystems/directories/files impacted by the change). They also capture process and developer history, such as the author's past development experience overall/in the recent time window/within the relevant subsystem, to better predict whether that commit will be defect-inducing.

However, LAPredict[8] challenges Kamei's expert features by relying solely on the LA (the number of lines added) feature. In their research, the authors tested different features and found that using only the LA feature can achieve high AUC and small performance variance within-project and cross-project. While other features like NUC (Number of Unique Changes) showed performance degradation when moving from within-project to cross-project test, the LA showed stability and resilience to variance.

The most important factor in JIT-SDP is the magnitude of change, even when the model architecture becomes more complex. DeepJIT [13] learns distributed vector representations of changes to the code using a Hierarchical Attention Network (HAN), supervised by log messages. CC2Vec [7] similarly learns representations of commit messages and code diffs using a Convolutional Neural Network (CNNs) to learn representations of the semantics of changes to the code beyond handcrafted features. However, studies continue to emphasize the importance of change magnitude as the most important predictor.

Beyond analyzing commit messages and code changes to predict defects, researchers have also explored separating LA and lines deleted (LD) into distinct categories. A challenge in accurately identifying defect-inducing commits is distinguishing between actual bug fixes and benign code modifications, such as refactoring. Refactoring, which restructures code without changing external behaviour, can be misclassified as buggy, particularly by models relying heavily on syntactic or structural features. Consequently, incorporating explicit refactoring analysis, such as methods derived from the Refactoring CAT approach [14], has been proposed as a way to improve the performance and precision of state-of-the-art JIT-SDP models by filtering out these noisy, non-buggy structural changes.

Existing research in expert features has predominantly focused on change magnitude rather than making more features by combining existing features, overlooking the magnitude and balance of code changes or a normalized Entropy (Distribution of modified code).

Name	Description	Dimension
NS	The number of subsystems affected by the change	Diffusion
ND	The number of directories impacted by the change	Diffusion
NF	The number of files altered by the change	Diffusion
ENT	Distribution of modified code across each file	Diffusion
LA	Lines of code added	Size
LD	Lines of code deleted	Size
LT	Lines of code in a file before the change	Size
FIX	Indicator of whether the change addresses a bug	Purpose
NDEV	The number of developers that changed the modified files	History
AGE	The average time interval between the last and current change	History
NUC	The number of unique changes to the modified files	History
EXP	Overall development experience of the author	Experience
REXP	Developer's experience in the recent time window	Experience
SEXP	Developer's experience within the relevant subsystem	Experience

Table 1. Studied 14 basic change-level features. Reproduced from [15]

2.2. **JIT-SDP Models**

2.2.1. **Models Based on Expert Features**

The field of JIT-SDP has evolved with the selection of traditional classifiers, DL and Pre-trained models (PTMs). LAPredict is an LR model solely depending on LA. By pairing this single metric with a traditional classifier, LR, researchers discovered that it could consistently outperform DL tools like CC2Vec [7] and DeepJIT [13] across 5 real-world projects. Beyond its accuracy, LAPredict is also efficient, proving to be 81k and 120k times faster in training and testing than CC2Vec and DeepJIT.

RF was implemented in SimCom [10] and JITLine [9]. Sim, the expert feature model from SimCom achieved a higher mean 0.043, 0.042, 0.064 than DeepJIT [13], the model automatically generating features using CNNs, in ROC, Precision, and F1-Score, respectively. It also achieved a higher mean 0.044, 0.044, 0.066 than CC2Vec [7], the model learning the relationship between actual code changes and semantics of the code changes using HAN, in ROC, Precision, and F1-Score, respectively, indicating a simple classifier can outperform DL. JITLine also implements RF and compares its performance against CC2Vec. The results show that RF achieves superior performance, improving PCI@20%LOC by 0.50–0.64. PCI@20%LOC measures the proportion of actual defect-inducing commits identified when inspecting only 20% of the total lines of code. The models with RF also have higher efficiency; JITLine (36–175 seconds across projects) runs 2–4 times faster than CC2Vec. However, JITLine and Sim did not compare its performance to the tuned XGBoost.

CodeBERT [16] is a transformer-based encoder-decoder model pre-trained on vast amounts of source code. When compared with RF (SimCom) and DL models like CC2Vec and CodeT5 demonstrated a higher computational cost while achieving lower predictive performance. This finding suggests exploring other classifiers trained solely on expert features.

2.2.2. **Models Based on Expert Features and Semantic Information**

To bridge the gap between simplicity and semantic expressiveness, current trends have shifted toward two main directions. The first integrates defect prediction with defect localization, to not only identify whether a commit is defective but also pinpoint where the defect resides. The second combines PTMs, which capture deep semantic information, with specialized models trained on expert features, to leverage both contextual understanding and expert features.

Most advancements, JITFine [5] and JITSmart [4], utilize feature fusion to combine the domain-specific insights of expert features with the deep contextual representations provided

by PTMs. JITFine integrates these features before tokenization to capture syntactic structures alongside traditional metrics, while JITSmart employs a multi-task learning framework to simultaneously predict defects and locate their origins. Comparative evaluations demonstrate that JITSmart generally achieves better performance, reaching a 1.1% higher F1-Score, a 0.05% higher AUC, and a 0.1% higher Recall@20%Effort than JITFine. JITSmart suggests MLP with a 0.3/0.7 loss function weight as a strong predictor. Therefore, newly suggested models can be compared to JITSmart.

We are exploring models for expert features that can be combined with semantic features. For semantic feature analysis, CodeBERT was used in several papers [4, 5]; however, it was having token limitation. JIT-Align [6] was introduced to solve this problem by using Euclidean distance and FAISS. FAISS is a database using PTMs to generate vector representations of commit messages and file-level code changes. By comparing the commit message and code message using Euclidean distance, F1-Score increased by 3.1%-9.6%, and MCC increased of 3.1%-9.7% across 5 open-source projects.

2.3. **Imbalanced Data**

To address the inherent imbalance in JIT-SDP datasets, JITLine [9] applies SMOTE optimized using Differential Evolution (DE) to generate synthetic minority samples. Meanwhile, DeepJIT [13], JITAlign, and LAPredict [8] focus on F1-Score and Matthews Correlation Coefficient (MCC) that are robust to class imbalance, rather than relying on Precision.

3. **Experimentation**

3.1. **Research Questions**

We investigated the following research questions for studying XGBoost:

> **(RQ1)** How much improvement can we achieve with feature engineering, HP tuning, imbalance-aware sampling, and weighting strategies in XGBoost?

For this RQ, we explore whether size-related expert feature engineering and HP tuning improve XGBoost. Imbalance-aware strategies (SMOTE and class weighting) are discussed to improve Recall and true positive rate for defective commits. We evaluate performance changes relative to a baseline XGBoost trained on Kamei's 14 expert features.

> **(RQ2)** Which classifier, LAPredict with LR, RF, or XGBoost, achieves the highest performance when trained solely on expert features?

For this RQ, the tuned XGBoost was compared to two other popular traditional classifiers. LR was tested only with LA, as LAPredict proves that LR performs best with LA alone.

> **(RQ3)** To what extent does XGBoost outperform MLP?

Although JITSmart employs an MLP that integrates both semantic and expert features, our comparison isolates the expert feature setting to ensure a fair evaluation of classifiers. The final comparison report will show the relative improvement per project to determine whether XGBoost consistently delivers meaningful gains.

3.2. **Dataset**

We utilized a large-scale dataset comprising five prominent open-source projects: Gerrit, JDT, Platform, and QT, provided by LAPredict [8]. The data spans from January 1, 2011, to

December 1, 2020. The defect-inducing commits were determined using the SZZ algorithms by tracing back from bug-fixing commits to the original code modifications that introduced the faults. LAPreidct use chronological 80/20 split to follow the same training/testing data partitioning of the original DeepJIT and CC2Vec work.

Project	#Changes	% Defect	Language
Qt	95,758	15.16	C++
OpenStack	66,065	31.68	C++
JDT	13,348	41.20	Java
Platform	39,365	37.74	Java
Gerrit	34,610	8.64	Java

Table 2. Summary of Projects Used in the Study

The dataset includes projects written in two programming languages, C++ and Java, for a total of 579,146 commits. The defect rate represents the proportion of buggy commits among all commits and ranges from 8.64% to 41.20%, indicating a strong class imbalance typical of real-world JIT-SDP datasets.

3.3. **Expert Features**

Kamei proposes 14 expert features for measuring commit risk, and they are categorized into five dimensions: **1) Diffusion:** NS (subsystems), ND (directories), NF (files), ENT (entropy of changes), **2)Size:** LA (lines added), LD (lines deleted), LT (lines modified), **3)Purpose:** FIX (indicator of a bug), **4)History:** NDEV (developer count), AGE (project age), NUC (unique changes) **5)Experience:** EXP (author experience), REXP (recent author experience), SEXP (author experience in subsystem), as shown in table 1.

3.3.1. **Feature Engineering and Feature Importance**

To analyze the contribution of individual features, we examine the importance of characteristics derived from the trained models, as illustrated in Figure 1. The results show that several baseline features, including lines added (`la`), number of unique changes (`nuc`), the number of developers that change the modified files (`ndev`), the number of modified files (`nf`), and the number of lines of code in a file before the change (`lt` is substantially higher than other features.

The predominance of size and history-related features suggests that the model is highly sensitive to the magnitude of the changes, while normalized and churn-related features were not addressed. To address this limitation, we propose additional engineered features to provide additional information that may not be explicitly covered by the original feature set. All the engineered features are prefixed with `eng_`.

Diffusion-based features. We introduce a normalized entropy feature to mitigate the bias introduced by large commits:

$$\texttt{eng\_norm\_entropy} = \frac{\texttt{entropy}}{\log(1 + \texttt{nf})}$$

This normalization reduces the tendency of entropy to scale directly with the number of modified files.

Size-based features. To explicitly characterize the magnitude and balance of code changes, we added the following features:

$$\texttt{eng\_churn\_ratio} = \frac{\texttt{la} + \texttt{ld}}{\max(1, \texttt{lt})}$$

$$\texttt{eng\_net\_churn} = \texttt{la} - \texttt{ld}$$

$$\texttt{eng\_abs\_churn} = \texttt{la} + \texttt{ld}$$

These features summarize both the total number of changes made and the balance between additions and deletions.

History-based features. We also introduce a feature on developer activity, `eng_NRDEV`, which is defined by the number of unique developers who made changes to the repository within the time period.

The relative change features were also investigated; these included the proportion of lines that are added, deleted, and churned at the file level. However, these features could not be computed due to the unavailability of historical file-level data.

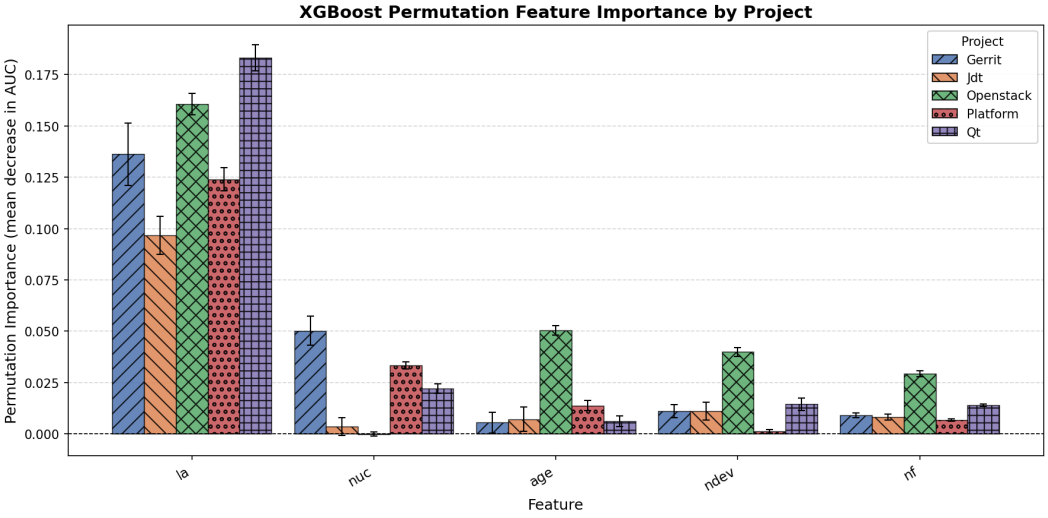

Figure 1. Grouped Bar Chart with Error Bars to compare feature importance.

3.4. **Hyperparameter Tuning**

XGBoost's hyperparameters can influence model performance, such as the tree depth, regularization terms, and sampling ratios. To mitigate overfitting, this study tunes the learning rate (`learning_rate` $\in [0.01, 0.3]$), minimum child weight (`min_child_weight` $\in [1, 10]$), and sampling ratios (`subsample` $\in [0.5, 1.0]$ and `colsample_bytree` $\in [0.5, 1.0]$). Moreover, we optimize structural complexity and regularization via the number of parallel trees (`num_parallel_tree` $\in [1, 5]$) and the gamma (`gamma` $\in [0.0, 5.0]$).

3.5. **Evaluation metrics**

AUC–ROC. AUC-ROC measures the model's ability to separate buggy commits from clean ones. Instead of picking a single cutoff threshold, it evaluates the model across all possible thresholds. Formally, AUC represents the probability that a randomly chosen true defective or clean commit is ranked higher than a randomly chosen negative defective or clean commit.

F1-score. F1-score evaluates the balance between precision and Recall, which is commonly used for imbalanced JIT-SDP datasets.

Brier score. Brier score measures the accuracy by computing the mean squared difference between predicted probabilities and actual outcomes:

$$\text{Brier} = \frac{1}{N}\sum_{i=1}^{N}(p_i - y_i)^2$$

where p_i is the predicted probability and $y_i \in \{0, 1\}$ is the true label.

Recall. Recall measures the model's ability to identify the defective commits, reflecting how often the model misses defects.

3.6. Experiment Environment

The experiments in this paper were conducted on an Apple M3 chip with 24GB of unified memory, running on macOS.

4. Experiment Results and Analysis

Hypothesis. To investigate the practical effectiveness of LAPredict, we analyzed its performance using a confusion matrix. The results reveal a limitation: the model consistently failed to predict buggy commits (defective) instances correctly. As shown in Figure 2, the vast majority of defective commits were misclassified as clean. This indicates that while LAPredict achieve a high AUC (mean 0.7246 within the project and cross projects) by ranking extreme outliers correctly, it fails to identify the defective commit that doesn't have an exceptionally high number of LA. The root cause of this failure lies in the combination of LR and imbalanced data. In JIT-SDP datasets, clean commits outnumber buggy ones. When trained on such data, the LR learns a conservative decision boundary.

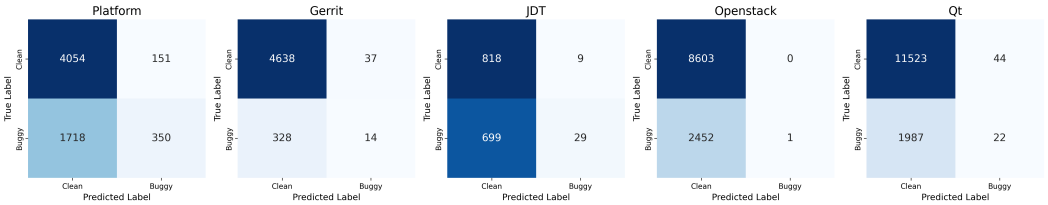

Figure 2. Confusion matrix of LAPredict using a Logistic Regression classifier.

The confusion matrix shows that only 14–350 defective commits were correctly identified (true positives), while 328–1718 defective commits were misclassified as clean (false negatives). This results in buggy recall between 0.04% and 4.1%, indicating that the majority of defects remain undetected. The primary goal of JIT-SDP is to catch as many buggy commits as possible before they are integrated into the codebase. A model with high precision but low Recall, like LAPredict, in many scenarios fails to provide the SDP that developers need. To prevent this, we added Recall as an evaluation metric.

Finding 1. Despite achieving a competitive mean AUC of 0.7246 across within- and cross-project settings, LAPredict exhibits extremely low buggy recall (0.04%–4.1%), failing to detect the majority of defective commits and limiting its practical applicability in JIT-SDP.

Several prior studies have reported that traditional classifiers can outperform more complex ML approaches while also being more cost-effective [5, 8]. Although LR and RF have been widely used in JIT-SDP, XGBoost was not compared to LAPredict, MLP(JITSmart) or RF(JITFine and Sim). Therefore, we experiment with XGBoost, as prior work indicates that it achieves superior performance in terms of average ranking compared to RF (2.07%) and LR (2.29%) [12].

Furthermore, LAPredict suggests that increasing the amount of data or the number of features does not necessarily lead to improved predictive performance. This observation

motivates further research on the effectiveness of traditional classifiers, as they need smaller datasets than DL.

4.1. **Engineered Feature Importance**

The top five features with the highest mean importance values, along with the corresponding standard deviations for XGBoost, are provided in Table 3. As shown in the results, LA consistently yields the highest values of importance across the five different projects, highlighting the dominant role of change size in JIT-SDP. Among the engineered features, eng_ABS_CHURN appears within the top-ranked features, suggesting that explicit modelling of total change volume provides additional predictive value. In contrast, eng_NRDEV is shown to have unstable values of importance and may actually negatively impact the model. This indicates that using coarse-grained developer diffusion information may not be beneficial.

Feature	Importance Mean	Importance Std	Dimension
Lines of code added	0.0948	0.0364	Size
The number of unique changes	0.0187	0.0133	History
The average time interval	0.0169	0.0205	History
Overall development experience of the author	0.0137	0.0155	Experience
Magnitude of code changes (eng_ABS_CHURN)	0.0125	0.0093	Diffusion

Table 3. Top 5 Feature Importance and Dimension

The above results suggest that feature engineering can improve the representation of size-related features. However, JIT-SDP models are the most strongly influenced by change magnitude, with an importance score of 0.0948.

4.2. **Impact of HP Tuning on Performance**

We performed HP tuning for XGBoost using Optuna. The tuned hyperparameters vary depending on the number of selected features (original Kamei's 14 features, top 5 features, original and engineered features, excluding `eng_NRDEV`) and whether class imbalance handling using SVMSMOTE is used.

HP tuning improves performance the most when applied to the top five features, suggesting a high degree of sensitivity to model configuration within reduced feature selection. The model with the top five features performs at a comparable level to the tuned XGBoost model with original features, suggesting that carefully selected features can be effective as using more features.

Finding 2. The performance using only the top five engineered features is comparable to the full feature set, with AUC-ROC differences ranging from -0.0277 to $+0.011$ and Recall differences ranging from -0.0706 to $+0.2864$. This demonstrates that carefully selected features could preserve their performance while reducing model complexity.

4.3. **An Optimized SMOTE Technique Impact on Performance**

To mitigate class imbalance and improve the true positive rate for defective commits, we apply SVMSMOTE exclusively to the training data. Prior studies have shown that the SMOTE technique outperforms other class rebalancing techniques [12, 17]. Among the SMOTE variants, SVMSMOTE was selected because it uses support vector machines to generate synthetic minority samples near the decision boundary and has been reported as the best sampling-based method in comparative evaluations [17].

Features	SMOTE	Metric	QT	OP	JDT	Plat.	Gerrit
Original	No	AUC-ROC	0.7291 (+.0084)	0.7448 (+.0118)	0.6886 (+.0212)	0.7918 (+.0102)	0.7286 (+.0305)
	No	F1-Score	0.3900 (+.0088)	0.4792 (+.0042)	0.6252 (+.0101)	0.6186 (+.0169)	0.2331 (+.0253)
	No	Recall	0.5336 (+.0000)	0.5809 (+.0008)	0.6552 (+.0110)	0.7108 (+.0280)	0.2778 (+.0205)
Top 5	No	AUC-ROC	0.7401 (+.0321)	0.7340 (+.0115)	0.6512 (+.0418)	0.7641 (+.0441)	0.7290 (+.0326)
	No	F1-Score	0.3864 (+.0129)	0.4755 (+.0053)	0.5976 (+.0400)	0.5744 (+.0277)	0.2347 (+.0200)
	No	Recall	0.6665 (+.0931)	0.6437 (+.0175)	0.6140 (+.0618)	0.6402 (-.0078)	0.5643 (+.2047)
Top 5	Yes	AUC-ROC	0.7407 (+.0006)	0.7248 (-.0092)	0.6439 (-.0073)	0.7646 (+.0005)	0.7297 (+.0007)
	Yes	F1-Score	0.3891 (+.0027)	0.4780 (+.0025)	0.6070 (+.0094)	0.5788 (+.0044)	0.2420 (+.0073)
	Yes	Recall	0.5630 (-.1035)	0.6519 (+.0082)	0.6387 (+.0247)	0.6499 (+.0097)	0.4298 (-.1345)
OG + Eng	Yes	AUC-ROC	0.7416 (+.0171)	0.7417 (+.0080)	0.7002 (+.0334)	0.7978 (+.0084)	0.7433 (+.0213)
	Yes	F1-Score	0.3737 (+.0032)	0.4643 (+.0185)	0.6614 (+.0073)	0.6189 (-.0005)	0.2273 (-.0105)
	Yes	Recall	0.6386 (-.0085)	0.7901 (-.0334)	0.9258 (+.0906)	0.9197 (-.0029)	0.3070 (+.00)

Table 4. Performance Comparison Before/After HP tuning

In our experiments, weighting the minority class with ratios such as 1:2, 1:20, and 1:30 often dropped the AUC-ROC by around 0.2 and did not yield effective improvements. Furthermore, SVMSMOTE produced almost identical results when using the top five features in 4. However, when combining original and engineered features, it improved AUC-ROC by 0.0025–0.0163 and Recall by 0.1052–0.2899 compared to the original

Finding 3. SVMSMOTE is particularly effective when applied to larger feature sets, suggesting that features with low individual importance benefit from imbalance-aware sampling. SVMSMOTE yields improvements of 0.0025–0.0163 in AUC-ROC and 0.1052–0.289 in Recall.

5. Discussion

RQ1: First, HP tuning was performed using the 14 features, resulting in improvements of 0.84%-3.05% in AUC-ROC in Table 4. Next, HP tuning was applied to the top 5 features, yielding the improvements of 1.15%-4.18%. Subsequently, HP tuning on the top 5 features with SVMSMOTE showed only marginal changes in performance (±0.01). The final tuned XGBoost with engineered features, HP tuning, and SVMSMOTE shows improvements of 1.19%-6.48% in AUC-ROC and 19.32%-43.70% in Recall, while keeping Brier score under 0.3 in Table 6.

Dataset	Metric	Default	Tuned	Change (%)
Platform	AUC-ROC	0.7816	0.7978	+2.07%
	F1-Score	0.6017	0.6189	+2.86%
	Recall	0.6828	0.9197	+34.70%
OpenStack	AUC-ROC	0.7330	0.7417	+1.19%
	F1-Score	0.4750	0.4643	-2.25%
	Recall	0.5801	0.7901	+36.20%
QT	AUC-ROC	0.7207	0.7416	+2.90%
	F1-Score	0.3812	0.3954	+3.73%
	Recall	0.5336	0.6386	+19.67%
JDT	AUC-ROC	0.6674	0.7002	+4.91%
	F1-Score	0.6151	0.6614	+7.53%
	Recall	0.6442	0.9258	+43.70%
Gerrit	AUC-ROC	0.6981	0.7433	+6.48%
	F1-Score	0.2078	0.2273	+9.39%
	Recall	0.2573	0.3070	+19.32%

Table 5. Performance Changes After Tuning

Tool	Metric	QT	OP	JDT	Plat.	Gerrit	Mean
XGBoost	AUC-ROC	0.7416	0.7417	0.7002	0.7978	0.7433	**0.7442**
	F1-Score	0.3954	0.4643	0.6614	0.6189	0.2273	**0.4747**
	Brier	0.1963	0.2669	0.2879	0.2399	0.1013	0.2185
	Recall	0.6386	0.7901	0.9258	0.9197	0.3070	**0.7099**
RF (JITFine, Sim)	AUC-ROC	0.7450	0.6955	0.7138	0.7800	0.7412	0.7351
	F1-Score	0.3417	0.6298	0.2071	0.6193	0.4869	0.4570
	Brier	0.1757	0.2855	0.1204	0.2895	0.2520	0.2246
	Recall	0.3091	0.6415	0.2485	0.7210	0.5809	0.5002
LAPredict (LR)	AUC-ROC	0.7409	0.6695	0.7318	0.7296	0.7503	0.7244
	F1-Score	0.0212	0.0757	0.0712	0.2725	0.0008	0.0883
	Brier	0.1243	0.2486	0.0652	0.2173	0.2010	**0.1713**
	Recall	0.0110	0.0398	0.0409	0.1692	0.0004	0.0523
MLP	AUC-ROC	0.7119	0.6918	0.6502	0.7604	0.7002	0.7029
	F1-score	0.3769	0.4294	0.6049	0.5972	0.2443	0.4505
	Brier	0.1592	0.1983	0.2429	0.1990	0.1445	0.1888
	Recall	0.4530	0.5393	0.6456	0.7162	0.4708	0.5650

Table 6. Performance comparison across projects.

RQ2: We ran 3 models (Tuned XGBoost, RF and LR) for 5 projects each and calculated their mean in Table 6. The final XGBoost resulted in the best AUC-ROC (0.7442) and F1-Score (0.4747), which are respectively 1.24% and 3.87% higher than the second-highest model, RF. Most importantly, the tuned XGBoost model achieved the highest mean recall (0.7099), which is 25.65% higher than that of MLP (0.5650). However, its Brier score was higher than those of LR and MLP, but lower than that of RF. The Brier score experiment proves that XGBoost is more accurate than RF in predicting defective commits.

RQ3: XGBoost consistently outperformed MLP, with improvements of 5.55% in AUC-ROC, 5.10% in F1-score, and 20.41% in Recall in Table 7. Moreover, XGBoost required, on average, 15.9 times faster training time than MLP. This indicates that XGBoost is more computationally efficient and accurate than MLP.

Model	Metric	QT	OP	JDT	Plat.	Gerrit	Mean
XGBoost	Training Time (s)	0.18	0.17	0.11	0.13	0.13	**0.144**
	AUC-ROC	0.7416	0.7417	0.7002	0.7978	0.7433	**0.7442**
	F1-score	0.3954	0.4643	0.6614	0.6189	0.2273	**0.4747**
	Recall	0.6386	0.7901	0.9258	0.9197	0.3070	**0.7099**
MLP	Training Time (s)	2.35	2.33	2.25	2.24	2.32	2.298
	AUC-ROC	0.7119	0.6918	0.6502	0.7604	0.7002	0.7029
	F1-score	0.3769	0.4294	0.6049	0.5972	0.2443	0.4505
	Recall	0.4530	0.5393	0.6456	0.7162	0.4708	0.5650

Table 7. Performance Comparison Between XGBoost and MLP

6. Threats to Validity

Internal Validity. While we tuned several critical hyperparameters, we did not conduct a grid or random search over the full parameter space. However, our tuning strategy prioritized mitigating overfitting by constraining model complexity and adding randomness to improve robustness to noise. Moreover, although feature engineering and selection were based on importance analysis and prior studies, we did not try other features than Kamei's expert features. Different features capturing different data may yield different performance outcomes, potentially influencing the comparative results.

External Validity. The study evaluation was conducted on five open-source projects, written in C++ and Java. Although these projects are adopted benchmarks in other JIT-SDP research, they do not fully represent other programming languages. Further replication studies should be conducted across additional languages and larger datasets.

Construct Validity. The study did not evaluate all possible performance metrics. However, we prioritized Recall, while ensuring that Brier score remained low and that AUC-ROC and F1-score were maintained at competitive levels. Although these metrics are widely used in JIT-SDP research, they may not fully capture all dimensions of practical defect prediction performance, such as effort-aware evaluation.

7. Conclusion

This study demonstrates that increasing the number of features does not necessarily improve predictive performance. Using only the top five most important features in combination with SMOTE yielded results comparable (± 0.03 in AUC-ROC and ± 0.3 in Recall) to those obtained with the full feature. The combination of engineered features, SVMSMOTE and HP tuning enhanced XGBoost performance, showing gains of 1.19%-6.48% in AUC-ROC and 19.32%-43.70% in Recall. Across the five open-source projects, the tuned XGBoost ourperformed LR, RF and MLP, achieving higher recall (25.65%-1257.94%), proving that XGBoost mitigate the low-recall limitation observed in LAPredict [8]. Moreover, XGBoost maintained higher AUC-ROC (1.24%-5.88%), and improved F1-Score (3.87%-437.60%). The smaller improvement of AUC-ROC and F1-Score compared to Recall shows that XGBoost catch more buggy commits, but also false positives (clean commits).

In future work, XGBoost should be integrated with code semantic features. Since XGBoost cannot directly learn semantic information from the code, a hybrid model using late fusion [10] may improve its performance. In addition, more data and software systems could be used to validate these findings.

References

[1] Y. Zhao, K. Damevski, and H. Chen. "A Systematic Survey of Just-in-Time Software Defect Prediction". In: *ACM Computing Surveys* (2024).

[2] F. Lomio, L. Pascarella, F. Palomba, and V. Lenarduzzi. "Regularity or Anomaly? On The Use of Anomaly Detection for Fine-Grained JIT Defect Prediction". In: *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2022, pp. 270–273.

[3] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, and S. Li. "Just-In-Time Defect Identification and Localization: A Two-Phase Framework". In: *IEEE Transactions on Software Engineering* 48.1 (2022), pp. 82–101.

[4] X. Chen, F. Xu, Y. Huang, N. Zhang, and Z. Zheng. "JIT-Smart: A Multi-task Learning Framework for Just-in-Time Defect Prediction and Localization". In: *Proc. ACM Softw. Eng.* 1.FSE (July 2024).

[5] C. Ni, W. Wang, K. Yang, X. Xia, K. Liu, and D. Lo. "The best of both worlds: integrating semantic features with expert features for defect prediction and localization". In: *Proc. 30th ACM ESEC/FSE*. 2022, 672–683. ISBN: 9781450394130.

[6] Y. Ye, H. Yu, G. Fan, Y. Liang, J. Dong, and W. Chen. "JIT-Align: A Semantic Alignment-Based Ranking Framework for Just-In-Time Defect Prediction". In: *2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC)*. 2025, pp. 1328–1337.

[7] T. Hoang, H. Kang, D. Lo, and J. Lawall. "CC2Vec: distributed representations of code changes". In: *Proc. ACM/IEEE 42nd ICSE*. June 2020, pp. 518–529.

[8] Z. Zeng, Y. Zhang, H. Zhang, and L. Zhang. "Deep just-in-time defect prediction: how far are we?" In: *Proc. 30th ACM SIGSOFT ISSTA*. ISSTA 2021. 2021, 427–438. ISBN: 9781450384599.

[9] C. Pornprasit and C. Tantithamthavorn. "JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction". In: *Proc. MSR*. 2021.

[10] X. Zhou, D. Han, and D. Lo. "Bridging expert knowledge with deep learning techniques for just-in-time defect prediction". In: *Empirical Software Engineering* 30.1 (2024).

[11] R. Shwartz-Ziv and A. Armon. "Tabular Data: Deep Learning is Not All You Need". In: *CoRR* abs/2106.03253 (2021). arXiv: 2106.03253.

[12] J. Bryan and P. Moriano. "Graph-Based Machine Learning Improves Just-in-Time Defect Prediction". In: *CoRR* abs/2110.05371 (2021). arXiv: 2110.05371.

[13] T. Hoang, H. Khanh Dam, Y. Kamei, D. Lo, and N. Ubayashi. "DeepJIT: An End-to-End Deep Learning Framework for Just-in-Time Defect Prediction". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, pp. 34–45.

[14] F. Niu, J. Shao, C. Mayr-Dorn, L. Huang, W. K. G. Assunção, C. Li, J. Ge, and A. Egyed. "Refactoring Bug-Inducing: Improving Defect Prediction with Code Change Tactics Analysis". In: *IEEE International Symposium on Software Reliability Engineering (ISSRE)*. 2025.

[15] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. "A large-scale empirical study of just-in-time quality assurance". In: *IEEE Transactions on Software Engineering* 39.6 (2013), pp. 757–773.

[16] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. arXiv: 2002.08155 [cs.CL].

[17] R. van Dinter, C. Catal, G. Giray, and B. Tekinerdogan. "Just-in-time defect prediction for mobile applications: using shallow or deep learning?" In: *Software Quality Journal* 31 (June 2023), pp. 1–22.