Journal of Software: Evolution and Process

WILEY

Software: Evolution and Process

RESEARCH ARTICLE - EMPIRICAL

Enhancing Feature Quality in JIT-SDP: An Empirical Study of Branch Dependency Mining

Zuowei Chen | Liyan Song

Faculty of Computing, Harbin Institute of Technology, Harbin, China

Correspondence: Liyan Song (songly@hit.edu.cn)

Received: 18 September 2025 | **Revised:** 13 June 2026 | **Accepted:** 30 June 2026

Keywords: branch dependency | feature noise | high-quality data | just-in-time software defect prediction

ABSTRACT

Just-in-time software defect prediction (JIT-SDP) aims to predict defect-inducing software changes in a timely manner, thereby enhancing development efficiency and product quality, especially during the software maintenance phase. Previous studies have primarily focused on proposing novel methods to achieve stronger predictive performance, often evaluated using open-source projects produced through data extraction tools. This extraction process typically operates under the implicit assumption that the extracted data features are exempt from noise. However, limited attention has been given to data quality, particularly regarding feature quality. We have identified that common feature extraction methods, such as those implemented by verb—Commit Guru—usually overlook branch dependency information, which can compromise feature quality and lead to unreliable or invalid conclusions. This paper systematically evaluates the quality of features extracted in JIT-SDP and investigates their impact on predictive performance. Moreover, we propose a novel feature extraction method that accounts for branch dependency to produce high-quality features. Our experiments based on 22 open projects reveal that neglecting branch dependency can significantly affect feature quality; however, this issue has a limited impact on the predictive performance of JIT-SDP models. Our results reveal that, although neglecting branch dependency introduces measurable feature discrepancies, the downstream predictive performance and model rankings of JIT-SDP models remain largely stable. These findings provide strong evidence regarding the empirical robustness of prior JIT-SDP studies, suggesting that the field's foundational conclusions are not compromised by this specific data quality issue. Our enhanced feature extraction method is publicly available at https://github.com/HongJinSecond/Feature-study.

1 | Introduction

Just-in-time software defect prediction (JIT-SDP) focuses on predicting defects at the code change level, enabling developers to identify potential defects promptly in the software development and maintenance process [1–4]. Recently, JIT-SDP has gained increasing attention from both academia [5] and industry [6–8], serving as a decision support tool to enhance software quality.

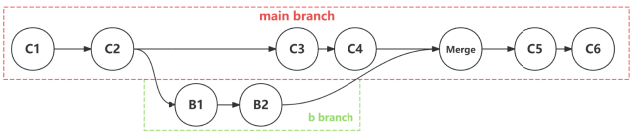
From a machine learning perspective, JIT-SDP can be framed as a binary classification task, where defect-inducing changes are classified as Class 1 and clean changes as Class 0 [9, 10]. Given the sequential nature of software changes and the frequent occurrence of concept drift [11, 12], JIT-SDP is suited to be treated as an online learning problem, where models are continuously updated with incoming data [9, 11, 13]. Numerous JIT-SDP methods have been proposed, primarily based on 14 expert features presented by Kamei et al. [4]. `Commit Guru` [14] is a widely used tool for extracting these 14 characteristics for each code change, which subsequently form the training and testing data streams used in related research to evaluate proposed methods.

Previous studies have primarily focused on proposing novel classifiers to enhance prediction performance [15–18]. These efforts usually operate under the implicit assumption that the extracted data features are free from noise. The quality of the datasets used for experimental validation is of significant importance, as poor data quality can undermine the performance validity of prediction models and, consequently, the reliability of conclusions drawn—such as which predictors yield superior performance.

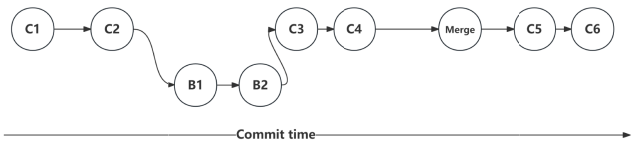
Existing feature extraction techniques, such as those implemented in `Commit Guru` [14], often neglect branch structure information. This oversight means that these methods focus solely on the chronological context of the target data when extracting features, ignoring pertinent predecessor and successor code changes that may originate from different branches.

Figure 1 illustrates the potential issues arising from neglecting branch structure. In Figure 1a, the true dependencies between code changes are presented, where C_2 and C_3 denote two code changes, with C_2 as the parent of C_3 on the "main" branch. The code changes B_1 and B_2, located on the "b" branch, are committed chronologically between C_2 and C_3. The "b" branch may have been created by another developer to implement a new member function, which will eventually be merged back into the "main" branch upon completion. However, when only chronological information is considered, ignoring the branch structure, as shown in Figure 1b, the code changes B_1 and B_2 would be incorrectly placed between C_2 and C_3 based solely on their chronological order. This misplacement results in the erroneous conclusion that B_2 is the parent of C_3, when in fact, C_2 is the true parent node.

Furthermore, Figure 1 highlights the importance of accurately identifying the correct parent by demonstrating the computation of feature LT (lines of code in a file before the change). For change C_2, the LT value should be derived from the lines of code associated with its true parent, C_1. However, if branch dependencies are overlooked, commit B_1 would be erroneously considered the parent of C_2. This misidentification would lead to using lines of code from B_1 instead of C_1 in the computation, leading to an inaccurate feature value for LT. This scenario illustrates



(a) Actual relationships between code commits with branch dependency.



(b) Erroneous relationships caused by missing branch dependency.

FIGURE 1 | An illustrative example demonstrating the necessity of incorporating branch dependency information into the feature extraction process.

that when the computation of feature values relies on correctly identifying the parent code change, overlooking branch dependency can result in noisy features. Therefore, neglecting branch dependencies and relying solely on the chronological order of code commits can result in inaccurate relationships between code changes, ultimately introducing unexpected feature noise. A more comprehensive examination and discussion of this issue can be found in Section 3.

This paper aims to systematically investigate whether branch-oblivious feature extraction threatens the validity and reliability of empirical conclusions in JIT-SDP research. To this end, we conduct a large-scale empirical study to quantify the feature discrepancies introduced by missing branch dependency information and evaluate their downstream impact on predictive performance and model ranking stability.

Moreover, we propose a novel method for extracting high-quality features by especially integrating branch dependency into the feature extraction process, thereby facilitating more reliable model training.

Unlike prior studies that primarily focus on improving predictive performance, our goal is not merely to design a more accurate feature extraction algorithm. Instead, we seek to understand whether a long-standing assumption in JIT-SDP data mining pipelines—namely, treating chronological dependency as sufficient for feature extraction—materially affects the validity, reproducibility, and robustness of empirical findings in the literature.

We conclude this study by answering the following research questions (RQs):

RQ1. To what extent does the neglect of branch dependency during feature extraction influence the accuracy of feature values?

RQ2. Given the potential significant impact on extracted feature values, how do noisy features affect the predictive performance of JIT-SDP models?

RQ3. If feature noise significantly impacts performance, what implications does this have for the conclusions regarding the relative rankings of various JIT-SDP models?

RQ1 serves as the foundation for the subsequent RQs by examining the impact of feature noise on feature values. If the answer to RQ1 is "no," we can conclude that neglecting branch dependency information has no effect on the extracted feature values and, consequently, no influence on the findings of previous studies. Conversely, if the answer is "yes," we must further investigate how this feature influences the model's predictive performance (RQ2). RQ3 acts as a crucial extension of RQ2 by analyzing the differences in model rankings—an aspect that researchers typically prioritize when proposing improved prediction models and evaluating predictive performance of JIT-SDP models.

To systematically address these three sequentially linked RQs, we design a four-stage evaluation pipeline and present it as a

high-level roadmap in Figure 2. The pipeline proceeds from data extraction and feature generation (using the conventional chronological method and our proposed branch dependency mining for feature extraction [BDFE]), through BDFE validation—comprising feature-quality verification against expert inspection and an execution-overhead assessment—and feature-noise quantification (RQ1), to an analysis of how this noise affects model predictive performance (RQ2) and model ranking (RQ3). This roadmap can serve as a guide to the overall workflow before the technical details that follow.

Our experiments based on 22 open-source projects show that feature noise resulting from the neglect of branch dependency affects the accuracy of five features (LT, NDEV, AGE, NUC, and REXP). However, this inaccuracy has minimal impact on model performance, as indicated by small effect sizes, and does not change the conclusions regarding the superiority of the prediction models. Notably, the rankings of JIT-SDP models remain consistent, suggesting that previous conclusions still hold.

The main contributions of this paper are as follows:

- To the best of our knowledge, we are the first to identify and quantify branch dependency in Git's revision graph as a distinct, structural source of feature noise in JIT-SDP, showing that ignoring it can corrupt a substantial fraction of the extracted feature values.
- We propose BDFE, a scalable, branch-aware extraction method that mines the revision graph to compute noise-free feature values, and we release BDFE together with the resulting datasets as a public replication package to support reproducibility.
- Using BDFE to construct a controlled comparison between the noisy and branch-consistent feature spaces, we conduct a large-scale empirical assessment across 22 open-source projects and five state-of-the-art (SOTA) JIT-SDP models, and find that although the feature discrepancies are statistically detectable, predictive performance and model rankings remain largely stable—thereby supporting the validity of prior JIT-SDP findings.

The remainder of this paper is organized as follows: Section 2 presents background and related work. Section 3 provides the motivation for this study. Section 4 proposes a novel feature extraction method for JIT-SDP. Section 5 outlines the experimental setup, including the datasets used, the metrics for assessing feature noise, and the benchmark prediction models. Section 6 discusses experimental results and presents answers to our RQs. Threats to validity are discussed in Section 8, and the paper is concluded in Section 9.

2 | Background and Related Work

2.1 | JIT-SDP

JIT-SDP is a special case of SDP operated at a finer code change level, which aims to predict whether software changes may induce defects [6, 8]. JIT-SDP can be modeled as a binary classification task, where a prediction model is constructed based on training examples of software changes that are produced with labels of *defect-inducing* (Class 1) or *clean* (Class 0). Considering the chronology of software changes [9] and the frequent changes in the generation of defective software changes [13, 19], JIT-SDP has been widely recognized to be modeled in an online learning framework [9, 16, 18], where additional software changes are produced and labeled over time and then used to update prediction models.

Prior JIT-SDP studies have primarily focused on proposing novel prediction models based on code change metrics to enhance predictive performance [12, 18, 20]. Online oversampling bagging (OOB) [21], initially designed to address online imbalance in machine learning, was introduced to the JIT-SDP by Cabral et al. [11] and has since been adopted as a benchmark in many subsequent studies [16, 17, 19, 22].

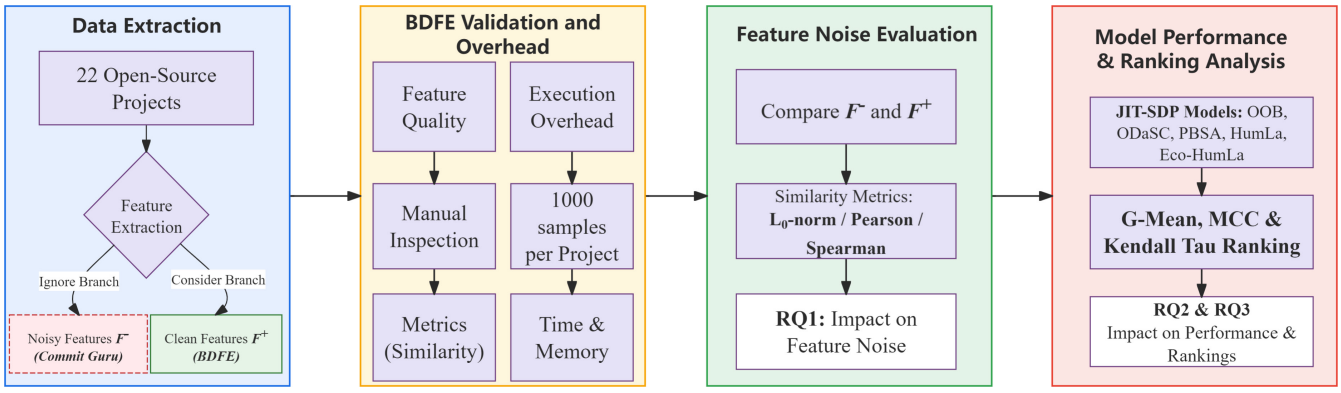


FIGURE 2 | Roadmap of the study. From 22 open-source projects, features are extracted using two methods: the chronological tool Commit Guru, which ignores branch dependency and yields the noisy set F^-, and the proposed BDFE method (Section 4), which accounts for branch dependency and produces the clean set F^+. BDFE is validated against expert manual inspection and its execution overhead is profiled (Section 5). The two feature sets are then compared via l_0-norm, Pearson, and Spearman metrics to quantify feature noise (RQ1, Section 6.1). Finally, F^- and F^+ are used to train five state-of-the-art JIT-SDP models; predictive performance is measured by G-Mean and MCC, and model rankings are compared using Kendall's τ, to evaluate the impact of noise on predictive performance (RQ2) and model ranking (RQ3) (Sections 6.2–6.3).

Later, to especially cater for the frequent drifting class imbalance in JIT-SDP, oversampling rate boosting (ORB) [11] was proposed based on OOB, dynamically adjusting the resampling ratio to accommodate changing class imbalances. Building upon ORB, prediction-based sampling adjustment (PBSA) [12] further incorporated prediction results into the resampling process to help the model adapt to concept drift. To mitigate one-sided label noise, Song et al. proposed oversampling based data stream bagging with confidence (ODaSC) [16], which uses adaptive micro-clusters to assess label confidence and correct labels of software changes. Recently, Song et al. [18] underlined human involvement in the defect prediction process, proposing HumLa (immediate human labeling for software changes predicted as defect-inducing) and Eco-HumLa (effort-conserve HumLa). When a new software change is predicted as defect-inducing, the developer is notified to review the change, facilitating timely updates to the training data and determining whether it indeed produces any defects. In this sense, a labeled training example can be created more promptly and used for model update in a more timely way. There are also studies that focus on some of the underlying factors in software changes, such as the chronological dependencies between changes [9] and human interference in actual development [18]. Because most models can only work on the project which they were trained on, researchers also want their model to be more general and can work well on other projects. Such models can be referred to as cross-project models [9, 20, 23].

TABLE 1 | Software change features [4].

Dim.	Metric	Description
Diffusion	NS	Number of modified subsystems.
	ND	Number of modified directories.
	NF	Number of modified files.
	Entropy	Distribution of modified code across each file.
Size	LA	Lines of code added.
	LD	Lines of code deleted.
	LT	Lines of code in a file before the change.
Purpose	FIX	Whether or not the change is defect fix.
History	NDEV	The number of developers that changed the modified files.
	AGE	The average time interval between the last and the current change.
	NUC	The number of unique changes to the modified files.
Experience	EXP	Developer experience.
	REXP	Recent developer experience.
	SEXP	Developer experience on a subsystem.

Many JIT-SDP methods rely on feature metrics summarized in Table 1, identified by Kamei et al. [4] as good indicators for defect prediction in their large-scale empirical study. They examined 14 features extracted from commits and bug reports, categorizing them into five dimensions: diffusion, size, purpose, history, and experiences. Their findings confirmed that these features serve as good indicators of high performance in JIT-SDP models. Many subsequent studies, including those focusing on online learning scenarios, have built upon these features [13, 15, 17, 20, 24].

2.2 | Data Quality in JIT-SDP

Mining and exploring high-quality data (features and labels) related to code changes in a software project is crucial and plays a significant role in software quality assurance. This practice can also greatly influence the results and conclusions of related research.

Several studies have focused on evaluating and improving datasets for JIT-SDP by enhancing the quality of labels [4, 25–30]. The majority of the related literature emphasizes improving auto-labeling quality when producing labels of software changes, particularly through the SZZ algorithm [25] and its variants. For instance, AG-SZZ [26] addresses issues related to empty lines and annotations, whereas the original SZZ algorithm incorrectly considers these text modifications as code changes, resulting in inaccurate labels. MA-SZZ [28] ignores modifications to document and text files, whereas the original SZZ treats all files as code files. RA-SZZ [27] is specifically designed for JAVA projects and can effectively capture code refactoring.

However, few studies have investigated the quality of data features [30], especially the expert features initially summarized by Kamei et al. [4], which are produced for change-level defect prediction. Herbold et al. primarily focus on feature selection and extension [30]. They compared the effectiveness of various features for defect prediction and provided a dataset containing a large number of features. Their findings indicate that the features most influential to SDP are those related to code churn, whereas the absence or addition of other features had minimal impact on performance. However, their study did not address the effects of feature noise, particularly in terms of data value dimensions.

Our preliminary investigation reveals that the current feature extraction methods often overlook the branch dependency of code changes, potentially resulting in feature noise (see Section 1). However, it remains unclear how this neglect affects the quality of data features created from code changes and how such feature noise influences the validity of predictive performance in JIT-SDP. Consequently, this paper aims to enhance the feature extraction process for JIT-SDP by incorporating branch dependency, thereby producing more reliable features for code changes. By comparing these improved features with the original ones, we will quantify and assess the significance of the discrepancies in feature noise. Furthermore, we will investigate the impact of this feature noise on the predictive performance

of JIT-SDP and evaluate the validity of conclusions drawn in prior research.

2.3 | Semantic Features

The 14 features inspected in this study are conventionally used in JIT-SDP studies and are commonly referred to as *expert features*, as they are typically proposed and hand-crafted by domain experts. These features encapsulate abstract information about code changes, making them suitable for predicting tasks in JIT-SDP.

In addition to expert features, researchers have started to explore extracting *semantic features* from code text, particularly with the emergence of deep learning [31–33] and large language models (LLMs) [34–38]. These features are generally derived through grammatical or lexical analysis, followed by preprocessing with word embedding models, which have demonstrated promising performance [32, 34, 37]. Beyond just code text, researchers have also considered additional textual information, such as commit message [31, 32, 34]. Rather than analyzing the entire codebase, studies [31, 32, 34] have focused on the code change lines. Recent studies [31, 34] have explored techniques for detecting defects at a more fine-grained level, such as by file or code block, achieved through the positioning of the text lines or word embeddings.

Despite that, expert features remain highly relevant and prevalent, especially in online learning scenarios [9, 17, 18], which is the main focus of this study. Exploring semantic features lies beyond the scope of this study, but represents a promising direction for our future studies. This work is significant in advancing the understanding of online JIT-SDP.

In practical applications, both types of features can be complementary and enhance predictive performance. Recent studies have tried to combine these features to further improve model effectiveness [33, 34, 36]. For instance, Ni et al. [34] combined expert features with code embeddings to create new input vectors for a linear classifier. They utilized the expert features produced by up-scaling networks alongside code vectors encoded by an encoder-only model like BERT. This combined approach yielded better performance than using either expert feature or semantic features alone.

3 | Motivation

As overlooking branch dependency leads to incorrect calculation of critical features, this discrepancy constitutes a non-negligible source of feature noise that threatens the internal validity of empirical studies in JIT-SDP. Prior work has largely relied on tools such as Commit Guru under the implicit assumption that the extracted features are accurate [4, 10, 12, 15, 18]. Yet whether this noise actually biases the evaluation and comparison of predictors remains an *open question* that cannot be answered without an accurate, branch-aware extraction method to serve as a reference. This motivates our proposal of BDFE (Section 4), which eliminates this specific, structural source of noise and thereby provides the *gold-standard* feature values needed both to reconstruct a reliable benchmark and to rigorously test whether the noise affects existing conclusions.

To illustrate the possible feature noise in JIT-SDP, we conducted a toy experiment designed to highlight the absence of branch dependency and the resulting parent–child issues. Using **Git** for project management, we created various branches and commits. For each commit, we randomly added or removed random lines of text to simulate typical software development processes. The number of lines added (LA) and removed (LD) per commit is documented in Table 2. Figure 1 displays the true branch structure of the toy project, where code changes B_1 and B_2 are located on a separate branch, which is subsequently merged into the "main" branch after code change C_4.

Table 2 reports the feature values for LA, LD, LT, and NUC, with their meanings provided in Table 1. LA and LD were chosen as typical features that are less vulnerable to feature noise associated with the absence of branch dependency, as their values are directly derived from committed code changes, making them less likely to be influenced by the misidentification of parental

TABLE 2 | Feature values for LA, LD, LT, and NUC for the toy project shown in Figure 1b, highlighting the significance of accurately determining parent–child relationships. The symbol "+" corresponds to BDFE, and "–" corresponds to `Commit Guru`. The middle column is the actual value calculated manually (LA and LD are unaffected by label noise stemming from the absence of branch structure; therefore, their values are not duplicated for conciseness).

Change	LA–	LA	LA+	LD–	LD	LD+	LT–	LT	LT+	NUC–	NUC	NUC+
C_1	1			0			0	0	0	0	0	0
C_2	2			1			1	1	1	1	1	1
B_1	3			1			2	2	2	2	2	2
B_2	4			1			4	4	4	3	3	3
C_3	3			1			7	2	2	4	2	2
C_4	1			1			9	4	4	5	3	3
C_5	2			0			9	9	9	6	6	6
C_6	0			0			11	11	11	7	7	7

code changes. Conversely, LT and NUC are more vulnerable to inaccuracy arising from the misidentification of parental code changes, which is the reason they were included in this toy experimentation.

The true feature values are indicated as LA, LD, LT and NUC, whereas the extracted features that disregard branch dependency are denoted as LA–, LD–, LT–, and NUC–. Features that account for branch dependency are marked as LA+, LD+, LT+, and NUC+ (as proposed shortly in Section 4).

As shown in Table 2, the features LT and NUC for code changes C_3 and C_4 on the "main" branch are affected when branch dependency is overlooked. In contrast, the values for earlier code changes that precede branch "b" (C_1 and C_2) and those that follow the merge of the two branches remain unaffected by this issue. Meanwhile, the feature values for LA and LD are consistent across all code changes, as expected, demonstrating identical values between LA and LA–, as well as between LD and LD–.

4 | BDFE

This section proposes our feature extraction method, which accounts for branch dependency by leveraging a revision graph in Git, termed *BDFE*. Our method consists of three main steps:

1. Mining branch dependency (Section 4.1): This step involves constructing a descriptive revision graph that captures branch and version information pertained to all code changes within the software project.
2. Incorporating branch dependency into feature extraction (Section 4.2): producing and maintaining data objects (*Code Change Logs*) that enable accurate feature calculation.
3. Enhancing memory efficiency (Section 4.3): a memory recycling strategy that reduces overhead and accelerates processing.

Together, these three steps produce the accurate, branch-aware feature values that underpin the rest of the study: the experimental setup in Section 5 uses these values to define and investigate our RQs on feature noise and its impact on predictive performance and model ranking (RQ1–RQ3), with the results reported in Section 6. The overall framework is illustrated in Figure 3.

4.1 | Mining Branch Dependency

The first step recovers the branch structure of a project's history, which GIT records only implicitly. We construct a *revision graph* in which nodes represent individual commits (code changes) and directed edges encode parent–child relationships. Building this graph is not entirely straightforward: although a commit's parents are readily available from the GIT LOG command, its children and the points where branches diverge are not recorded and must be inferred. Algorithm 1 details the construction procedure, and Table 3 defines the per-node properties in populates. We describe the process step by step below.

Using the data structure defined in Table 3, we execute the GIT LOG command to retrieve all code changes from the code repository. Each change corresponds to a graph node that records commit information, initially with empty property values (Line 3 of Algorithm 1). Next, we gather the properties listed in Table 3 for each node object, which will assist in producing the dependency structure later on. To facilitate quick access and enhance the efficiency of the search process, we employ a dictionary data structure to store the node objects corresponding to each software change.

Subsequently, we establish links between each node and its parent and child nodes, preserving this information within the dictionary H (Line 4 of Algorithm 1). Identifying the parent node(s) for a given code change is straightforward, as this information is provided by the GIT LOG command. However, determining the child node(s) is more complex, as this information is not explicitly recorded. We infer the parent–child relationships through the following iterative steps:

1. Assign the identified parent ID to the node property *Parent ID*.
2. Assign the *Commit ID* of the current code change to the node property *Child ID* of the identified parent node.

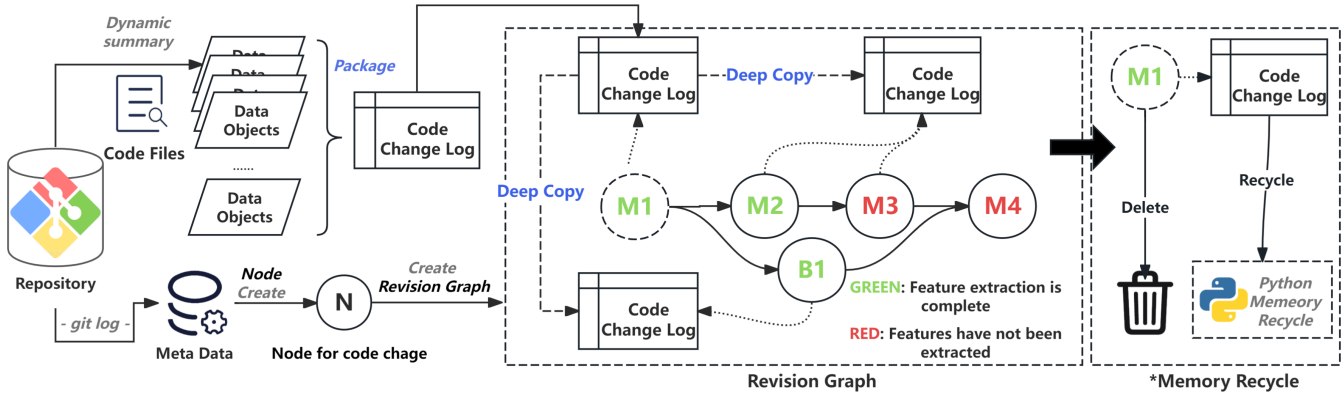


FIGURE 3 | BDFE: branch dependency mining for feature extraction.

TABLE 3 | Node structure associated with code change C.

Node property	Description
Commit ID	The unique identifier associated with C.
Parent ID(s)	The commit ID representing the parent code change of C. A commit may have multiple parent IDs.
Child ID(s)	The commit ID representing the child code changes of C. A commit may have multiple child IDs.
Code Change Log	A dictionary of data objects that records essential information, including the numbers of lines, the number of developers, and the last modified time for each file in the project.
New branch [Y/N]	Record whether this code change corresponds to the start of a new branch.
Recycled [Y/N]	Indicate whether this node can be recycled and its memory space released (used in Section 4.3).

3. Determine the node property *New Branch*, which indicates whether the parent node initiates a new branch. A node is considered the start of a new branch if it has more than one child node.

Upon completing this procedure, all commits of the software project will be interconnected, creating a comprehensive *revision graph*. This structure allows for easy access to branch and parent–child dependencies by checking the nodes in dictionary H, thus yielding an accurate revision graph.

4.2 | Incorporating Branch Dependency Into Feature Extraction

Algorithm 1. Mining branch dependency.

Input: Code changes $C_1, \cdots, C_n$

Output: Structured code changes with the data structure of dictionary H. Each code structure has the properties outlined in Table 3.

1: $H = \{\}$
2: **for** each C_i **do**
3: Create node N_i for change C_i.
4: Link N_i to its parent(s).
5: Insert N_i into H
6: **end for**
7: **return** H

Algorithm 2. Feature extraction and *Code Change Log* update.

Input: The revision graph dictionary H which is created by our Algorithm 1.

Input: The id i of one commit that we need to do feature extraction.

Input: The modification message m.

1: Load the node N from H based on i.
2: Load the node P from H based on the property "Parent ID(s)" of N.
3: **if** The property "New Branch" of P is Y **then**
4: Create a new *Code Change Log* for N which is a copy version from P.
5: **else**
6: N directly uses the same *Code Change Log* as P.
7: **end if**
8: Calculate the value of features for node N based on its *Code Change Log* and the message m.
9: Update the *Code Change Log* of N by m.
10: Check whether P and its *Code Change Log* can be recycled or not.

With the revision graph built, the second step computes branch-aware feature values. The key idea is that a commit's features depend on the historical state of the files it touches, which we capture in a per-node data structure called the *Code Change Log* (Table 3). As the graph is traversed, each node inherits its parent's *Code Change Log*—copied when a new branch diverges and merged when two branches join—so that every commit's features are derived from the correct, branch-specific file history. This traversal and the feature computation are formalized in Algorithm 2.

In real-world project development, it is worth noting that modifications made to a file on one branch do not affect files in other branches until a **git merge** is performed. Consequently, each branch maintains its own *Code Change Log*, containing individual modification information pertinent to that branch. Upon creating a merge commit, the *Code Change Logs* from the two branches are combined into a single *Log*.

As outlined in Algorithm 2, we employ the log information to update the data object for each modified file and calculate feature values based on this information. For each incoming commit of code change, we first load the node object from our revision graph dictionary H using the unique ID of the commit, designating it as node N. We then load the parent node of N as node P. To extract feature values for N, we require the modification information from this commit, which can be obtained from the message generated by the GIT LOG command. However, this information alone is insufficient; we also need a detailed historical modification for the files in the project. *Code Change Logs* serve as this historical modification summary, facilitating the feature extraction process.

Before extracting features for node N, its *Code Change Log* is initialized as empty and must inherit data from its parent node P. If P is the starting node of a new branch, such as M_1 in Figure 3, we first copy the *Code Change Log* of P, and then pass it on to its child node(s). Each branch maintains its own version of the *Code Change Log* to prevent conflicts until the branch is merged. Conversely, if P is a node like M_2 in Figure 3, which has only one child node N and is not the start of a new branch, N can simply use the same *Code Change Log* as P. Once we retrieve information from the constructed *Code Change Log* and the modification information m, we can use them to calculate feature values and update the *Code Change Log* accordingly. It's worth noting that the method for calculating a feature's value varies depending on the specific feature being analyzed.

A particular scenario arises when a commit node has two parent nodes. To accommodate this situation, as illustrated by node M_4 in Figure 3, we combine the *Code Change Logs* of M_3 and B_1 into a new *Log* for M_4. As previously mentioned, the *Code Change Log* contains information about modified files and if a file appears in both *Logs*, we aggregate the associated data into a new *Log*. Other file information is copied directly into the new *Code Change Log*.

4.3 | Enhancing Memory Efficiency

Having shown how feature values are computed from inherited *Code Change Logs*, we now address the practical cost of maintaining them. Because a large project may contain tens of thousands of commits across many branches, naively copying and retaining a full *Code Change Log* at every branch point would incur prohibitive memory overhead, undermining the efficiency of the method. To prevent this, the third step introduces a *memory reclamation* strategy that releases a node's *Code Change Log* as soon as all of its children have been processed, so that only the logs still required for ongoing feature extraction remain in memory.

To address this issue, we implement a memory reclamation technique. After the features of a node have been extracted, BDFE inspects its parent(s): if all of a parent's children have been processed, that parent's *Code Change Log* is no longer needed. At that point, the parent's `Recycled` property is set to Y, and the node, together with its *Code Change Log*, is released in the final step of Algorithm 2.

For example, as illustrated in Figure 4, the feature extraction task for node B_1 is completed (marked in green in the figure). The memory reclamation mechanism then accesses the parent node of B_1, i.e., node M_1 in the figure. At this stage, the program detects that both child nodes of M_1 (namely, M_2 and B_1) have completed feature extraction; consequently, it identifies M_1 as an invalid node and performs memory reclamation. Node M_1 is directly deleted from memory by the invoked program. Meanwhile, the corresponding *Code Change Log*of node M_1 calls the built-in memory reclamation script of Python to determine whether the node can be released. This operation significantly enhances processing speed and reduce memory overhead.

We have implemented our new feature extraction method and compared it with the tool `Commit Guru`. The results, shown in Table 2, demonstrates that our proposed BDFE method can extract more reliable feature values that closely align with the ground-truth.

5 | Experimental Setup

5.1 | Dataset Preparation

We selected 22 open-source projects from GitHub (see Table 4) to investigate the feature noise resulting from the absence of branch structure and to evaluate its impact on JIT-SDP model performance. Thirteen of these projects were adapted from Song et al. [18] and further updated with recent commits, whereas nine were newly collected for this study. The selection criteria included a minimum duration of 4 years, a rich commit history of 3000–170,000 commits (with a mean of approximately 50,000 commits), star counts exceeding 8000, and wide defect-inducing change ratios of ranging from 1% to around 50%. The projects span various domains, including game engines, development plugins, and AI frameworks.

Data collection was conducted using the tool `Commit Guru` [14], a well-established JIT-SDP data extraction tool that

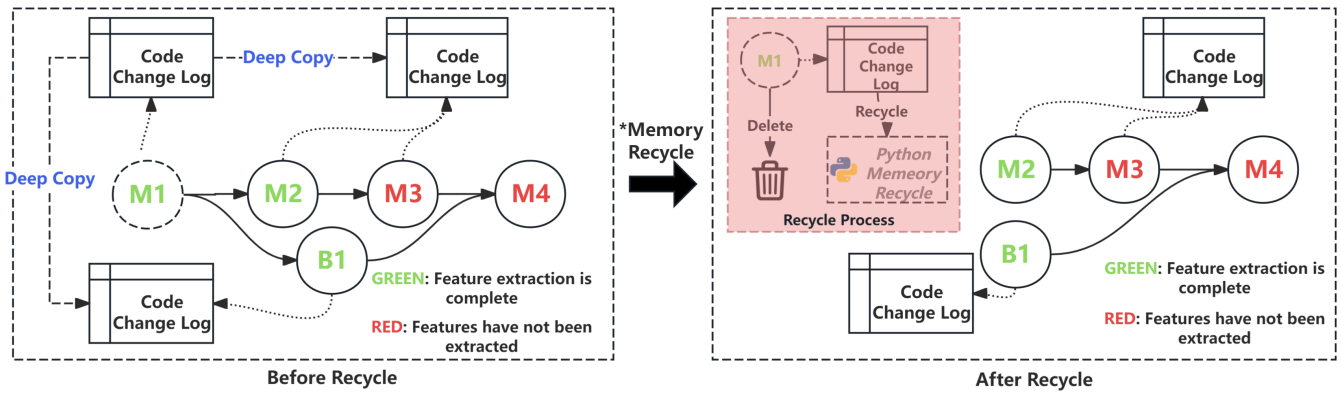


FIGURE 4 | Memory reclamation in BDFE. Each node retains its *Code Change Log (CCL)* only while an unprocessed child may still need it.

TABLE 4 | An overview of the projects. (#Changes denotes the total number of code changes, #Defects denotes the number of defect changes, Ratio% denotes the defect ratio, and "Stars" refers to the stars the project has gained on GitHub.)

Dataset	#Changes	#Defects	Ratio%	Stars	Time period	Language
Brackets	11,996	4116	34.31	33.3k	12/2011–03/2020	JavaScript
Broadleaf	14,602	3095	21.19	1.7k	12/2008–02/2024	Java
Camel	69,035	18,184	26.34	5.5k	03/2007–03/2024	Java
Fabric8	12,874	3289	25.55	1.8k	04/2011–03/2018	Java
Nova	32,001	15,674	48.98	3.1 K	05/2010–03/2024	Python
Tomcat	25,988	9482	36.48	7.5k	03/2006–03/2024	Java
Django	31,376	12,850	40.95	79k	07/2005–03/2024	Python
Rails	68,014	16,744	24.62	55.5k	11/2004–03/2024	JavaScript
Corefx	19,766	881	4.46	17.7k	11/2014–11/2019	C#
Rust	174,738	2040	1.16	96.5k	06/2010–03/2024	Rust
Tensorflow	146,655	33,649	22.94	185k	11/2015–02/2024	C++
Vscode	108,140	2391	2.21	162k	11/2015–03/2024	JavaScript
Wp-calypso	59,869	15,702	26.23	12.4k	01/2014–03/2024	JavaScript
Npm	8268	1828	22.11	17.5k	09/2009–07/2018	JavaScript
Node	40,108	12,335	30.75	106k	02/2009–03/2024	JavaScript
Pandas	31,138	16,997	54.58	43.2k	07/2009–03/2024	Python
Elasticsearch	70,402	25,618	36.38	69.4k	02/2010–03/2024	Java
Springboot	31,911	6251	19.59	74.4k	10/2012–03/2024	Java
Mybatis3	3699	841	22.74	19.7k	05/2010–03/2024	Java
Godot	33,903	13,004	38.36	88.6k	04/2013–03/2024	C++
Pytorch	69,887	29,504	42.22	81.8k	01/2012–03/2024	C++
Security	13,954	2653	19.01	8.7k	03/2004–03/2024	Java

implements the SZZ algorithm [25]. Following the methodological framework of prior JIT-SDP studies [18, 39, 40], we excluded commits from the last six months to ensure label quality, as it typically takes time for developers to confirm whether a commit is truly clean.

To investigate the impact of feature noise stemming from ignoring branch dependency, we produce two datasets for each project that share the same labels but may differ in feature values: one with feature noise and the other without. The first group of datasets was produced without considering branch dependency, following the approach used in `Commit Guru`. We refer to this feature space as $F^- \in \mathbb{R}^{n\times d}$, where n denotes the total number of software changes committed to the project, and d denotes the dimension of the created features. The second group of datasets incorporates branch dependency into the feature extraction process, as implemented in our proposed BDFE approach, leading to a feature space free from such feature noise, denoted as $F^+ \in \mathbb{R}^{n\times d}$. Similar to F^-, n denotes the total number of software changes for the project, and d denotes the feature dimension.

In total, we collected 44 datasets: 22 with noisy feature spaces arising from neglecting branch dependency and 22 that are free from such noise.

5.2 | Execution Overhead of BDFE

For developers and researchers, the computational cost of feature extraction is a practical concern, particularly because extraction is performed once, offline, to construct a dataset, rather than inside a latency-sensitive prediction loop. To quantify this cost, we compared the runtime and peak memory of BDFE against the standard chronological method. For each of the 22 projects, we extracted features for the same 1000 commits with both methods under identical hardware and software conditions. The experiments were performed on a virtual machine configured with 8 GB of RAM and a 300 GB SCSI hard disk, running Ubuntu 22.04 under VMware with a NAT network adapter. Using a fixed sample size across projects isolates the per-commit overhead and makes the figures comparable across repositories of very different sizes. The results are reported in Table 5.

TABLE 5 | Computational overhead of BDFE (new) versus the standard chronological method (old), measured over 1000 commits per project. `Diff` = `New`−`Old`; `Pct` (%) = (`Diff`/`Old`) × 100%. Positive `Diff` indicates an increase.

	Time (s)				Memory (MB)			
Project	**Old**	**New**	**Diff**	**Pct**	**Old**	**New**	**Diff**	**Pct**
Brackets	4.81	11.86	7.06	146.70	78.25	79.50	1.26	1.61
BroadleafCommerce	10.43	14.12	3.70	35.46	97.46	99.32	1.86	1.91
Camel	3.07	9.09	6.03	196.71	30.61	31.70	1.09	3.57
Corefx-v-6.2	5.80	22.23	16.43	283.40	49.77	51.21	1.44	2.89
Django	3.44	8.32	4.88	141.74	38.37	39.65	1.28	3.34
Elasticsearch	8.35	27.09	18.74	224.59	76.11	77.75	1.63	2.15
Fabric8	2.84	12.73	9.89	348.83	27.15	31.83	4.68	17.24
Godot	102.63	220.77	118.14	115.11	432.26	438.94	6.69	1.55
Mybatis3	1.12	2.79	1.67	148.24	14.57	14.43	-0.13	-0.92
Mysql-server	150.97	203.43	52.46	34.75	400.95	411.55	10.60	2.64
Node	33.22	49.63	16.41	49.39	200.36	202.52	2.17	1.08
Nova	29.79	48.83	19.04	63.93	170.66	175.32	4.67	2.74
Npm	2.58	4.86	2.29	88.82	44.39	45.53	1.14	2.56
Pandas	19.80	30.51	10.71	54.11	81.10	83.52	2.42	2.98
Pytorch	48.82	77.79	28.97	59.34	446.20	450.54	4.34	0.97
Rails	7.15	13.20	6.05	84.61	80.52	82.71	2.19	2.72
Rust	42.23	65.83	23.60	55.89	456.61	463.00	6.40	1.40
Security	5.94	9.49	3.55	59.72	78.53	79.98	1.45	1.84
Spring-boot	18.19	24.89	6.70	36.81	254.74	260.23	5.49	2.16
Tomcat	10.72	12.90	2.17	20.28	96.46	98.81	2.34	2.43
Vscode	51.91	108.47	56.56	108.96	621.03	629.26	8.23	1.33
Wp-calypso	24.81	75.36	50.55	203.75	410.71	415.85	5.13	1.25

Time overhead. BDFE increases extraction time by between $+20.28\%$ (Tomcat) and $+348.83\%$ (Fabric8), with a median increase of roughly $+87\%$ ($\approx 1.9\times$). In absolute terms, the additional time ranges from 2.17 s (Tomcat) to 118.14 s (Godot). The absolute overhead grows with project size, whereas the relative overhead is governed by the branching density of the repository: projects with frequent branching and merging, such as Fabric8, require more revision-graph traversal and therefore incur the largest relative slowdowns.

Memory overhead. In contrast, memory consumption is essentially unchanged: The relative increase is below 3% for 21 of the 22 projects, with a maximum absolute increase of 10.60 MB. The single outlier, Fabric8 ($+17.24\%$), is the same heavily branched project that shows the largest time overhead; its large percentage reflects a small baseline (27.15 MB) rather than a large absolute cost ($+4.68$ MB). The marginal decrease observed for Mybatis3 (-0.92%, -0.13 MB) is within measurement noise.

Why this cost profile is acceptable. The extra time stems from constructing the revision graph: because separate branches are materialized, some nodes are visited more than once. Crucially, this does not change the asymptotic complexity, as both the chronological baseline and BDFE (Algorithms 1 and 2 run in linear time $O(N)$; BDFE adds only a bounded constant factor that depends on the repository's branching structure, empirically between $1.2\times$ and $4.5\times$ in our data. Because the cost is linear in the number of commits, the overhead for a full project is predictable from these per-1000-commit measurements. One might expect the branch copies to inflate memory substantially; the fact that they do not is direct evidence for the memory-reclamation mechanism of Section 4.3, which releases revision-graph nodes as soon as all of their children have been processed.

In short, the overhead of BDFE is modest in the setting where it is actually used. Because feature extraction is a *one-time*, offline step, even the largest observed slowdown adds at most about two

minutes per 1000 commits and never materially affects downstream model training or evaluation. We therefore consider this cost well justified whenever feature accuracy or data consistency is a goal, and we discuss when that is the case in Section 7.1.

5.3 | Manual Validation of Feature Extraction Results

This section validates the correctness of the feature values produced by BDFE. To this end, we manually inspect a subset of commits to obtain human-verified ground-truth feature values, and compare them with the values extracted by Commit Guru and by BDFE. We emphasize that this manual inspection is a validation procedure only: it establishes a reference for assessing extraction accuracy and is not a qualitative analysis of the RQs, which are addressed entirely through the quantitative study in Section 6.

Noise-free features are essential for conducting experimental studies. However, manually reviewing all the tens of thousands of commits across 22 projects to verify feature correctness is impractical due to the sheer volume of code changes. Furthermore, obtaining accurate values for certain features presents inherent challenges: Frequent modifications to branches and paths by developers can obscure the true state of the software's evolution, whereas the large number of branches and substantial volume of commits further complicate this manual task. Consequently, the study requires an automated feature extraction approach that can provide sufficiently reliable feature values for large-scale analysis. In this subsection, we therefore evaluate whether the feature values produced by BDFE are sufficiently reliable for the subsequent experiments.

To validate the quality of the extracted features, we first prepare a sufficient number of commits, with their feature values meticulously inspected by experts to ensure they are free from noise associated with branch dependency. Specifically, we created a manual validation dataset by randomly selecting 100 commits per project for review. Six experienced developers, each with four years of development experience, were organized into three groups to independently inspect these samples. Their task was to determine whether branches had been forcibly modified or files renamed. For commits affected by non-standard branch operations, the corresponding feature values were manually checked and recalculated when necessary. The manually inspected results were used only as a validation reference for evaluating the correctness of the extracted feature values by our BDFE. Commits with inconsistent inspection results among the groups were excluded, resulting in a final validation dataset containing 1573 commits. This manually checked dataset is included in our replication package.

We will further substantiate our choice by comparing the similarities between the manually inspected feature values, our BDFE values, and the values from `Commit Guru`. The metric used to measure feature similarity is defined Equation (1). Higher similarity scores indicate greater consistency and reliability of the produced feature values relative to the ground-truth values. Table 6 presents the feature similarities, showing that our BDFE features outperform the `Commit Guru` features in terms of consistency with the ground-truth features, particularly regarding the NUC and NDEV features. These results indicate that BDFE can provide relatively reliable and noise-reduced feature values for the subsequent experimental studies. In the remainder of this paper, the features extracted by BDFE are denoted as F.

5.4 | Metrics for Evaluating Feature Value Similarity

This section provides metrics to measure the differences / similarity between each noisy feature values and their corresponding noise-free feature values. These measurements are used to jointly justify the BDFE features (Section 5.3) to answer RQ1 (Section 6.1).

For a given project, we denote all values of the m-th noisy feature as $F_m^- \in F^-$ and the corresponding noise-free feature as $F_m \in F$, where $m = 1, \cdots, d$. Using this notation, the discrepancy of the m-th noisy feature can be quantified by

$$\rho_m = ||F_m - F_m^-||,$$

where $||\cdot||$ represents a specific distance metric, such as the l_0 and l_2 norms. In this study, we implement the feature-wise discrepancy metric as follows: We first count the number of mismatches between each element of the two feature vectors F_m and F_m^-, and then compute the ratio of these discrepancies, formulated as follows:

$$\rho_0 = \frac{||F_m - F_m^-||_0}{n}, \tag{1}$$

where n denotes the number of software changes for this project and $||\cdot||_0$ is the l_0 norm for a vector. This metric measures the ratio of noisy values for each feature relative to the total number of code changes.

Moreover, we also adopt correlation metrics to evaluate the similarity between two groups of features, specifically Pearson correlation and Spearman correlation. Pearson correlation is employed to assess linear associations, whereas Spearman correlation, being robust to non-normality and monotonic

TABLE 6 | Feature similarity to the ground-truth values obtained through expert inspection. F^- represents feature values extracted by `Commit Guru` and F^+ denotes those extracted by our BDFE. The similarity metric is defined as $1 - \rho_0$ where ρ_0 is specified in Equation (1).

Datasets \ Features	LT	NDEV	AGE	NUC	REXP
F^-	0.58	0.84	0.71	0.85	0.68
F^+	0.64	1	0.77	1	0.74

nonlinearities, provides a complementary measure of general monotonic trends. For the m-th feature F_m, let x denote the values of its samples, with x_i representing the value of the i-th sample. We denote the samples of feature F_m^- as y, where y_i is the i-th sample. The mean values of x and y are represented as $\overline{X}$ and $\overline{Y}$, respectively.

Pearson correlation (denotes as r) measures the strength and direction of a linear relationship between two continuous variables. It evaluates how well the variation in one variable predicts linear variation in another [41]. The range of r is $[-1, 1]$, where $r = 1$ indicates a perfect positive linear correlation, $r = -1$ indicates a perfect negative linear correlation, and $r = 0$ suggests no linear correlation, although nonlinear relationships may still exist. The Pearson correlation can be formulated as follows:

$$r = \frac{\sum_{i=1}^{n}(x_i - \overline{X})(y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{X})^2}\sqrt{\sum_{i=1}^{n}(y_i - \overline{Y})^2}}. \tag{2}$$

Spearman correlation (denoted as r_s) assesses the strength and direction of a monotonic relationship (whether linear or nonlinear) between two variables. As a nonparametric method, it operates on the ranks of the data values rather than their raw magnitudes. The range of r_s is also [-1,1], where $r_s = 1$ indicates a perfect monotonic increasing relationship and $r_s = -1$ indicates a perfect monotonic decreasing relationship. Let $R(x_i)$ denote the rank of the i-th observation in variable X (a subset of F_m) and $R(y_i)$ denote the rank of the i-th observation in variable Y (a subset of F_m^-). The equation can be formulated as follows:

$$r_s = \frac{\sum_{i=1}^{n}(R(x_i) - \overline{R(X)})(R(y_i) - \overline{R(Y)})}{\sqrt{\sum_{i=1}^{n}(R(x_i) - \overline{R(X)})^2}\sqrt{\sum_{i=1}^{n}(R(y_i) - \overline{R(Y)})^2}}. \tag{3}$$

Furthermore, we employ statistical tests to assess the significance of feature noise between each pair of F_m and F_m^*. Wilcoxon signed-rank tests with a significance level of 0.05 [42] are used to evaluate statistical significance. The p-values calculated from the Wilcoxon tests will be further processed by Benjamini method [43]. Effect size A12 [44] is then utilized to measure the magnitude of significant differences, with larger values indicating greater impact on feature noise. This effect size is interpreted according to the categories established by Vargha and Delaney [44] as small (≥ 0.56), medium (≥ 0.64), and large (≥ 0.71).

5.5 | Benchmarking JIT-SDP Models

To answer RQ2 and RQ3, we will compare the predictive performance of JIT-SDP models constructed from the noisy feature space F^- and the noise-free feature space F. We have selected five SOTA JIT-SDP models for this analysis: OOB [21], ODaSC [16], PBSA [12], HumLa, and Eco-HumLa [18].

These prediction methods are specifically chosen for scenarios where JIT-SDP models are particularly applicable and effective in online learning context. OOB has been widely recognized as a benchmark model for JIT-SDP in prior studies, especially for its handling of the popular class imbalance problem presented in JIT-SDP datasets [11, 16, 22]. ODaSC [16] is selected for its ability to address label noise stemming from labeling delays. This method dynamically adapts to the temporal uncertainties associated with defect labeling as commits evolve over time. PBSA [12], an advanced version of ORB [11], is chosen for its sophisticated capability to adjust resampling rates over time. This adaptability enables it to effectively respond to the fluctuating class imbalance levels in evolving JIT-SDP data streams, thereby optimizing the learning process in real-time. HumLa and effort-conservative HumLa (Eco-HumLa), recently proposed by Song et al. [18], incorporate human inspection into the defect prediction process. By querying labels for potentially suspect instances, HumLa and Eco-HumLa can significantly enhance both practical applicability and prediction performance, acknowledging the resource constraints and decision-making processes inherent in real-world development settings.

5.6 | Predictive Performance Evaluation

We use the geometric mean of Recall 0 and Recall 1 (G-Mean) [45] to evaluate the predictive performance of the JIT-SDP models. Unlike accuracy or precision, G-Mean is known to be robust against class imbalance, making it particularly suitable for studies suffering from class imbalance evolution such as JIT-SDP [11, 15, 46]. We also employ the Matthews correlation coefficient (MCC) [47, 48], as it has gained popularity in the JIT-SDP domain [49–51].

For each dataset, we conduct 30 runs and report the average performance across these runs. G-Mean and MCC are computed sequentially using a *fading factor* to monitor changes in predictive performance over time, as recommended for the online learning scenarios [52]. The fading factor $\theta \in [0, 1]$ controls the emphasis placed on past evaluation examples compared with new ones, with larger/smaller values favoring historical/recent performance status. The fading factor $\theta = 0.99$ is used in this paper, following previous JIT-SDP studies [11, 16, 22], enabling a good trade-off between rapidly tracking performance changes and preventing excessive fluctuations. When computing average predictive performance, we use the mean of the sequential performance metrics.

To complete the answer to RQ3, we require a metric to quantify: (1) the ranking of JIT-SDP models built on the noisy feature space F^- and (2) the ranking of JIT-SDP models built on the noise-free feature space F. To achieve this, we employ Kendall's Tau [53], a well-established measure of ranking correlation that counts the numbers of concordant and discordant pairs to measure the similarity between two rankings. Specifically, for each feature space, we rank the models based on their performance, assigning rank scores accordingly. For example, if HumLa achieves the best performance in terms of MCC, it receives a score of 5, whereas the worst-performing model only receives a score of 1.

6 | Experimental Results and Analysis

6.1 | RQ1: Impact of Branch Dependency on Feature Noise

Table 7 presents the experimental results for RQ1, investigating the impact of missing branch dependency on the extraction process of noisy features, as measured by Equation (1). For clarity, features that are unaffected by this issue have been excluded from this table. The results show that five features (LT, NDEV, AGE, NUC, and REXP) are consistently influenced by branch-oblivious extraction, whereas the remaining features remain largely unaffected. This observation aligns with the findings of Kamei et al. [4], who noted that the affected features are calculated based on the most recent modification records of the files altered in the current commit. However, relying solely on chronological order and assuming that the most recent change is definitive contradicts branch management principles in software development, thereby compromising the validity of subsequent analyses.

Table 7 also reveals that the ratios of noisy feature entries vary across datasets, indicating that different projects experience varying degrees of feature noise. Notably, for the LT feature, Rust exhibits the highest level of feature noise, with nearly 61% of the feature values affected; whereas, Tomcat exhibits only 0.8% of affected feature entries for LT.

In addition, our analyses indicates that the impacts on features are inter-dependent: when a specific feature in a project is significantly affected, there is a higher likelihood that other features within this project will also exhibit substantial noise contamination. Some datasets, such as Brackets and Elasticsearch, demonstrate more severe feature noise, with noise ratios exceeding 30% across all five noisy features. Conversely, projects such as Tomcat and Django are less affected, with noise ratios below 1%.

Moreover, Tables 8 and 9 report Pearson and Spearman correlations, respectively, which measures the similarities between datasets F^- and F. These results indicate a relatively high linear relationship between the two datasets, suggesting that the feature noise cased by neglecting branch dependency does not significantly affect the overall data distribution. However, some projects still experience considerable impacts. For Pearson correlation, as shown in Table 8, SpringBoot and Rust exhibit low correlation coefficients, whereas the other projects display high coefficients. Table 9 reveals that a greater number of projects have low coefficients compared with Pearson correlation, including Brackets, Broadleaf, Elasticsearch, Godot, Vscode, Nova, and Tensorflow.

TABLE 7 | The noise ratio for each feature affected by the absence of branch dependency, calculated by Equation (1). This metric measures the proportion of noisy feature values over the total number code changes for each feature.

Datasets	LT	NDEV	AGE	NUC	REXP
Brackets	0.4249	0.3821	0.3119	0.4462	0.3028
Broadleaf	0.3091	0.2821	0.2467	0.3280	0.2441
Camel	0.0231	0.0235	0.0143	0.0253	0.0141
Corefx-v-6.2	0.2325	0.2245	0.2114	0.2454	0.2078
Django	0.0119	0.0104	0.0103	0.0125	0.0096
Elasticsearch	0.4071	0.4281	0.3566	0.4150	0.3491
Fabric8	0.0520	0.0392	0.0474	0.0590	0.0472
Godot	0.3517	0.3328	0.3559	0.3642	0.3381
Vscode	0.4943	0.8380	0.2155	0.5030	0.2115
Pytorch	0.0194	0.0432	0.0127	0.0206	0.0123
SpringBoot	0.3810	0.3833	0.2756	0.3959	0.2644
Node	0.0595	0.0510	0.0447	0.0623	0.0418
Nova	0.3746	0.3279	0.3380	0.3883	0.3304
Tomcat	0.0080	0.0010	0.0052	0.0084	0.0052
Pandas	0.0737	0.0534	0.0527	0.0766	0.0514
Security	0.2109	0.1628	0.1574	0.2529	0.1520
Rails	0.1852	0.1811	0.1596	0.1961	0.1502
Rust	0.6126	0.6957	0.4341	0.6265	0.4208
Wp-calypso	0.0528	0.0453	0.0412	0.0551	0.0405
Npm	0.0224	0.0231	0.0105	0.0254	0.0103
Mybatis3	0.0633	0.0600	0.0916	0.1092	0.0854
Tensorflow	0.3125	0.3187	0.2641	0.3184	0.2597

TABLE 8 | The Pearson correlation coefficient between the two datasets ($F+$ and F). The formulation for calculating this coefficient can be found in Equation (2).

Datasets	LT	NDEV	AGE	NUC	REXP
Brackets	0.8159	0.6332	0.8705	0.7678	0.9425
Broadleaf	0.7199	0.6741	0.7566	0.8571	0.9729
Camel	0.9983	0.9670	0.9872	0.9881	0.9942
Corefx-v-6.2	0.7034	0.8090	0.9086	0.9886	0.6983
Django	0.9933	0.9993	0.9977	0.9979	0.9998
Elasticsearch	0.9184	0.3528	0.5642	0.9055	0.7382
Fabric8	0.9957	0.9905	0.9858	0.9998	0.9988
Godot	0.9033	0.5474	0.7956	0.9254	0.9970
Vscode	0.8493	0.6680	0.7848	0.8005	0.9558
Pytorch	0.9983	0.9997	0.9923	0.9998	0.9999
SpringBoot	0.5976	0.3813	0.6278	0.5862	0.9988
Node	0.9844	0.9954	0.9882	0.9754	0.9996
Nova	0.8167	0.6152	0.8277	0.8089	0.9897
Tomcat	0.9998	0.9958	0.9987	0.9990	0.9985
Pandas	0.9672	0.9984	0.9891	0.9891	0.9987
Security	0.9238	0.7430	0.8259	0.9822	0.9508
Rails	0.8938	0.7779	0.8651	0.8556	0.9971
Rust	0.5974	0.2375	0.4709	0.6445	0.9141
Wp-calypso	0.9903	0.9866	0.9900	0.9971	0.9930
Npm	0.9994	0.9989	0.9984	0.9975	0.9996
Mybatis3	0.9331	0.9007	0.9348	0.9828	0.3788
Tensorflow	0.8704	0.7825	0.8904	0.8458	0.9936

To assess the statistical significance of the effect of feature noise, we further apply Wilcoxon signed-rank tests [42]. The null hypothesis (H0) states that feature values F_m and F_m^- are statistically similar, whereas the alternative hypothesis states that they differ significantly. The two-tailed pairwise Wilcoxon signed-rank tests show that all p-values for each dataset and each noisy feature are below the significant level of 0.05 after using Benjamini adjustment, leading us to reject H0. This indicates that the absence of branch dependency has a statistically significant impact on feature extraction for LT, NDEV, AGE, NUC, and REXP. The effect sizes, calculated using A12 [44], are categorized as small, with all values being less than 0.56. For example, for LT, the A12 effect size is 0.3482 for Vscode, and 0.3362 for Brackets; for NDEV, the A12 effect size is 0.1473 for Rust and 0.4987 for Pytorch.

Answer to RQ1: Among the 14 JIT-SDP features, five features (LT, NDEV, AGE, NUC, and REXP) exhibit measurable discrepancies when branch dependency is ignored during feature extraction. Although these discrepancies are statistically detectable, their practical impact remains limited according to the observed effect sizes.

6.2 | RQ2: Impact of Feature Noise on Predictive Performance

Table 10 reports the performance difference among models across 22 projects in terms of MCC and G-Mean. The results indicate that, despite the presence of feature discrepancies introduced by branch-oblivious extraction, the downstream predictive behavior of JIT-SDP models remains highly stable across projects.

Notably, most models perform better with the noise-free features, although with a few exceptions such as OOB on Bracket and Broadleaf projects. This also suggests that different models may be mildly affected differently depending on the specific project.

To further assess the consistency of these performance differences across projects, we employ Wilcoxon signed-rank tests together with the A12 effect size metric. The null hypothesis (H0) states that the performance differences are statistically similar, whereas the alternative hypothesis states that these differences are significantly different. The effect size is considered meaningful only if H0 is rejected, specifically when the p-value is less than 0.05. The statistical results presented in Table 11 reveal that the feature REXP is not significant in

TABLE 9 | The Spearman correlation coefficient between the two datasets ($F+$ and F). The formulation for calculating this coefficient can be found in Equation (3).

Features	LT	NDEV	AGE	NUC	REXP
Brackets	0.4965	0.4961	0.4533	0.4497	0.7788
Broadleaf	0.5596	0.5491	0.4862	0.5813	0.7863
Camel	0.9656	0.9556	0.9638	0.9740	0.9925
Corefx-v-6.2	0.6385	0.7947	0.6217	0.7006	0.7860
Django	0.9841	0.9900	0.9855	0.9828	0.9876
Elasticsearch	0.4958	0.1301	0.2709	0.4042	0.6845
Fabric8	0.9562	0.9941	0.9140	0.9588	0.9480
Godot	0.5512	0.4017	0.5175	0.4394	0.5709
Vscode	0.5715	0.5757	0.5552	0.5720	0.8172
Pytorch	0.9860	0.9994	0.9750	0.9904	0.9882
SpringBoot	0.4359	0.1889	0.3911	0.3893	0.7482
Node	0.9207	0.9925	0.9501	0.9171	0.9466
Nova	0.5587	0.4716	0.5959	0.5210	0.6982
Tomcat	0.9991	0.9965	0.9929	0.9979	0.9949
Pandas	0.8921	0.9927	0.9395	0.9158	0.9655
Security	0.8552	0.6650	0.6966	0.8141	0.8605
Rails	0.7506	0.6907	0.6859	0.7086	0.8801
Rust	0.3813	0.0250	0.2197	0.2714	0.7374
Wp-calypso	0.9227	0.9871	0.9404	0.9523	0.9491
Npm	0.9770	0.9823	0.9817	0.9782	0.9904
Mybatis3	0.9050	0.8691	0.8403	0.9105	0.9382
Tensorflow	0.5285	0.7796	0.5154	0.6156	0.7359

many projects, whereas the other features are nearly significant across all 22 projects. This finding indicates that the targeted features are effective, thereby having the potential reinforcing the validity of follow-up experimental results.

However, as shown in the last row of Table 10, the corresponding effect sizes (A12) are consistently small (all < 0.56). This suggests that although the noise is detectable by statistical tests (often due to large sample sizes), its practical impact on model efficacy is negligible, which demonstrates the robustness of SOTA JIT-SDP models against this specific type of input perturbation.

6.2.1 | Feature Level Explanation: Importance of the Affected Features

A first explanation is that the features most affected by branch dependency are not the ones that the models rely on most heavily. To examine this, we conduct a permutation-importance analysis [54], which quantifies the drop in predictive performance when each feature is randomly shuffled. We employ a random forest as a representative tree-based surrogate: it shares the decision-tree base learners of the five online-bagging models we evaluate, and permutation importance is largely *model-agnostic*, making its ranking a reasonable proxy. For each project we randomly sample 1000 balanced instances (500 defective, 500 clean) from F^+, repeat the procedure 30 times, and average the resulting scores.

Figure 5 reports the importance of the 14 features. All features contribute, but their importance is highly uneven. The most influential feature by a wide margin is LA, with a median importance of 0.09; critically, LA is *not* affected by branch dependency. Guo et al. report a consistent finding: code-churn metrics closely correlated with LA account for the largest share of defect-prediction effort [55]. Among the five features that are affected by branch dependency (LT, NDEV, AGE, NUC, REXP), four (LT, AGE, NUC, REXP) rank among the *least* important features overall. Only NDEV is comparable to the unaffected features, and even then its importance is marginally (their median importance ranges from 0.001 to 0.014).

This pattern explains both aspects of our results. Because the dominant predictive signal (LA) is computed correctly regardless of branch dependency, and because four of the five noisy

TABLE 10 | Differences in predictive performance between JIT-SDP models constructed using feature space F and those using feature space F^{-}. Positive (negative) values indicate better (worse) performance with noise-free features. The last row indicates the significance of these performance differences, with a slash ("/") denoting no significance difference. An effect size of less than 0.56 is considered as small.

	MCC					G-Mean				
Project	**OOB**	**ODaSC**	**PBSA**	**HumLa**	**Eco-HumLa**	**OOB**	**ODaSC**	**PBSA**	**HumLa**	**Eco-HumLa**
Brackets	0.0059	−0.0077	0.0209	−0.0090	−0.0148	0.0305	0.0095	0.0188	−0.0014	−0.0029
Broadleaf	0.0369	0.0171	0.0228	−0.0036	−0.0052	0.0597	0.0201	0.0348	−0.0043	−0.0035
Camel	−0.0015	0.0166	0.0007	−0.0016	0.0011	0.0010	0.0180	0.0020	−0.0009	0.0012
Corefx	−0.0702	0.0118	−0.0172	−0.0235	−0.0243	−0.1214	0.0088	0.0001	−0.0008	0.0002
Django	−0.0026	−0.0088	−0.0006	−0.0015	−0.0017	−0.0015	−0.0046	0.0003	−0.0012	−0.0010
Elasticsearch	0.0155	0.0079	0.0149	0.0035	−0.0035	0.0377	0.0067	0.0287	0.0000	−0.0052
Fabric8	0.0210	0.0150	0.0009	−0.0041	−0.0020	0.0539	0.0191	0.0020	−0.0024	−0.0011
Godot	0.0436	0.0187	0.0475	0.0034	0.0038	0.0875	0.0521	0.0983	−0.0017	0.0093
Vscode	−0.0052	0.0083	−0.0341	−0.0478	−0.0387	0.0034	0.0290	−0.0206	−0.0274	−0.0202
Pytorch	0.0224	0.0235	0.0095	−0.0033	−0.0007	0.0317	0.0170	0.0009	−0.0017	0.0019
SpringBoot	0.0461	0.0875	0.0505	0.0022	0.0036	0.0490	0.0926	0.0478	0.0005	0.0004
Node	0.0023	0.0254	0.0048	−0.0067	−0.0020	0.0118	0.0168	0.0136	−0.0095	−0.0058
Nova	0.0700	0.0284	0.0707	0.0014	−0.0048	0.1380	0.0236	0.1249	−0.0083	−0.0068
Tomcat	−0.0003	−0.0010	0.0004	0.0055	0.0076	0.0001	0.0002	0.0014	0.0050	0.0047
Pandas	−0.0002	−0.0154	0.0058	−0.0074	0.0043	−0.0002	−0.0116	0.0076	−0.0125	0.0046
Security	0.0013	0.0025	−0.0010	−0.0090	−0.0038	0.0028	0.0014	0.0020	−0.0042	−0.0024
Rails	0.0222	0.0414	0.0164	−0.0025	0.0055	0.0554	0.0876	0.0410	−0.0020	0.0097
Rust	−0.0424	0.0039	0.0311	0.0143	0.0279	−0.0804	0.0138	0.0945	0.0695	0.0968
Wp-calypso	0.0044	0.0236	−0.0001	−0.0034	−0.0071	0.0072	0.0292	0.0033	0.0071	0.0047
Npm	−0.0024	0.0095	0.0018	−0.0078	0.0048	0.0028	0.0028	−0.0009	−0.0027	0.0017
Mybatis3	−0.0035	−0.0221	−0.0020	−0.0074	−0.0190	0.0087	−0.0128	0.0099	−0.0101	−0.0117
Tensorflow	0.0208	−0.0052	0.0303	−0.0050	0.0039	0.0356	−0.0002	0.0304	−0.0080	0.0150
Effect-size	/	0.4607	0.4607	0.5268	/	0.3946	0.4111	0.4008	0.5247	/

TABLE 11 | The statistical significance of 14 features across 22 projects. We established that a *p*-value less than 0.05 indicates statistical significance, where "✓" represents significant features and "✗" represents none-significant ones.

Project	FIX	NS	ND	NF	ENT	LA	LD	LT	NDEV	AGE	NUC	EXP	REXP	SEXP
Brackets	✓	✓	✗	✓	✓	✗	✗	✗	✓	✓	✗	✗	✓	✗
Broadleaf	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	✓	✗	✓	✗
Camel	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
Corefx-v-6.2	✓	✓	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	✗	✓
Django	✓	✓	✗	✓	✓	✗	✗	✓	✓	✓	✓	✓	✗	✓
Elasticsearch	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✗	✓	✗	✓
Fabric8	✓	✓	✗	✗	✓	✓	✓	✗	✗	✓	✓	✓	✗	✗
Godot	✗	✓	✗	✓	✓	✗	✗	✓	✓	✓	✗	✓	✗	✗
Vscode	✗	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓
Pytorch	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Spring-boot	✓	✓	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✓	✓
Node	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗
Nova	✓	✗	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓
Tomcat	✓	✓	✓	✗	✓	✗	✗	✓	✓	✓	✓	✓	✗	✓
Pandas	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
Security	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✗	✓
Rails	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✗	✓
Rust	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
Wp-calypso	✓	✗	✗	✗	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
Npm	✓	✗	✓	✓	✓	✗	✗	✓	✓	✓	✓	✓	✗	✓
Mybatis3	✓	✗	✗	✗	✓	✗	✗	✗	✓	✓	✓	✗	✗	✗
Tensorflow	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

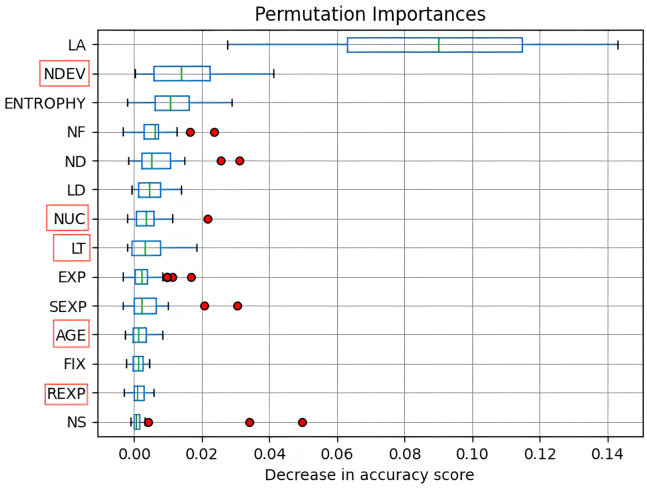


FIGURE 5 | Permutation importance of the 14 features, quantifying the decrease in JIT-SDP predictive performance when each feature is randomly shuffled.

features carry little weight, the overall effect of the noise on performance is small. At the same time, the moderate importance of NDEV is consistent with the small but statistically detectable effect reported for most models in RQ2. In short, the feature noise concentrates in features that models weight *lightly*, so its influence on predictions is *inherently limited*.

6.2.2 | Model-Level Explanation: Robustness to Feature Noise

Beyond the low permutation importance of the affected features (Section 6.2.1), a second explanation lies in the inherent *robustness* of the prediction models themselves. The five SOTA JIT-SDP methods evaluated in this study—OOB, ODaSC, PBSA, HumLa, and Eco-HumLa—are all built upon the online-bagging framework [56] with decision trees as their base learners. This shared architectural foundation endows them with two robustness properties that are directly relevant.

We emphasize that the robustness discussed below is an *explanation* of our observations under these representative, SOTA models, not a precondition we imposed in their selection. Conversely, this also means our conclusions are conditioned on this model family: architectures more sensitive to noise may behave differently, which is why we recommend BDFE as a safeguard in such settings (Section 7.1).

First, decision trees are resilient to feature noise because they partition the feature space through axis-aligned splits based on local thresholds, rather than relying on global optimization [57]. A small perturbation in a feature value is therefore *unlikely* to alter the hierarchical split path unless it happens to cross a split threshold, which sharply limits how far noise can propagate through the model.

Second, online bagging approximates traditional bagging [58] in a streaming setting while preserving its variance-reduction property. By aggregating predictions over many bootstrap samples, online bagging dilutes the influence of any individual noisy instance across the base learners, so isolated errors rarely change the ensemble's output.

Together, the trees' local decision boundaries and the ensemble's averaging makes these online-bagging-based JIT-SDP models naturally tolerant of the level of feature noise observed in our study. Combining with the low permutation importance of the affected features (Section 6.2.1), this *model-level* robustness explains why the feature noise introduced by neglecting branch dependency has only a *marginal* impact on predictive performance and leaves model rankings unchanged.

Answer to RQ2: Feature noise arising from neglecting branch dependency has a statistically significant effect on the predictive performance of most JIT-SDP models (with the exception of Eco-HumLa); however, this effect is consistently small in magnitude, as indicated by the small A_{12} effect sizes. We attribute this limited impact to two *complementary* factors: At the *feature*level, the affected features carry comparatively low permutation importance, whereas the dominant predictive feature (LA) remains unaffected (Section 6.2.1); and at the *model* level, the online-bagging tree ensembles evaluated here are inherently robust to small feature perturbations (Section 6.2.2).

6.3 | RQ3: Impact of Feature Noise on Model Ranking

Figure 6 presents the Kendall tau coefficients [53] for each performance metric across the 22 datasets. Although minor fluctuations in rank scores are observed (e.g., in PBSA), the median correlation coefficient exceeds 0.8, indicating strong consistency in model rankings. Crucially, the small effect sizes observed in RQ2 do not translate into significant changes in relative model performance. This ranking stability implies that the comparative conclusions of prior studies (e.g., which model is superior) remain valid despite the presence of feature noise. This robustness suggests that the practical utility of existing JIT-SDP tools is not critically compromised by this data quality issue.

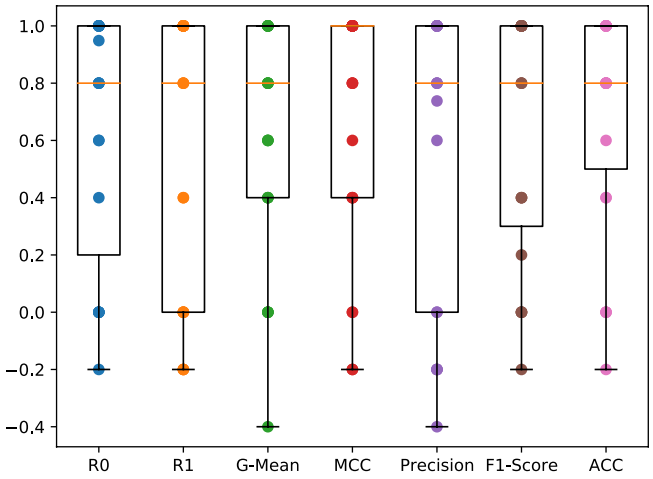


FIGURE 6 | Kendall tau correlation across all projects in terms of two primary metrics (G-Mean and MCC) along with five additional metrics (R0, R1, precision, F1-score, and accuracy).

Figure 7 presents the rank scores and the ranking comparisons for each JIT-SDP model across the datasets in terms of G-Mean and MCC, providing a more detailed view of ranking correlations. The results indicate that feature noise may influence the rank scores of certain SDP models, such as PBSA. Specifically, the rank score of PBSA is affected for both for G-Mean and MCC; however, the magnitude of this effect is minimal. For instance, the rank score of PBSA improves when employing a noise-free feature space F compared with noisy feature spaces F^- in terms of MCC, yet its relative ranking changes only from the the lowest to the second-lowest position.

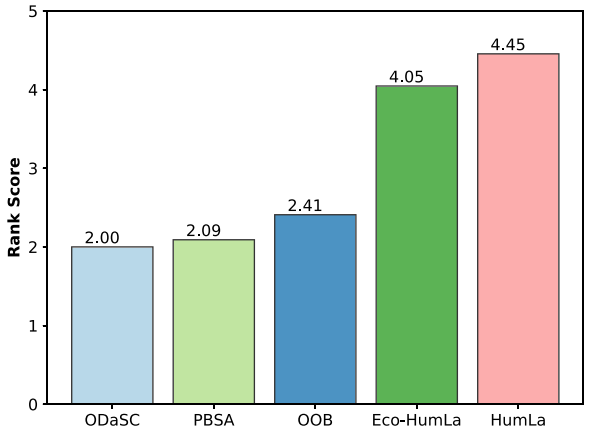
In terms of G-Mean and MCC, both HumLa and Eco-HumLa consistently perform the best, regardless of whether they are constructed using noisy or noise-free feature space. Across all datasets, the models with the lowest rank scores are consistently ODaSC, PBSA, and OOB. These results indicate that although the rank scores of the models may vary to some extent, the overall ranking structure remain stable. This stability suggests that the conclusions regarding the performance of the models are robust and reliable.

Answer to RQ3: Feature noise arising from the absence of branch dependency does not significantly impact the rankings of JIT-SDP models. This indicates that such feature noise is unlikely to compromise the validity and accuracy of prior academic studies.

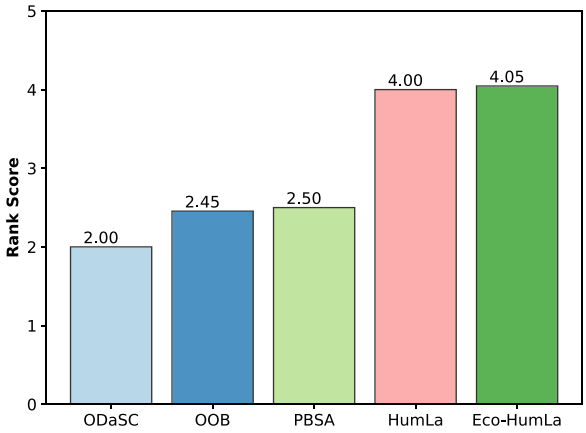
7 | Discussion

7.1 | Practical Implications: When to Use BDFE

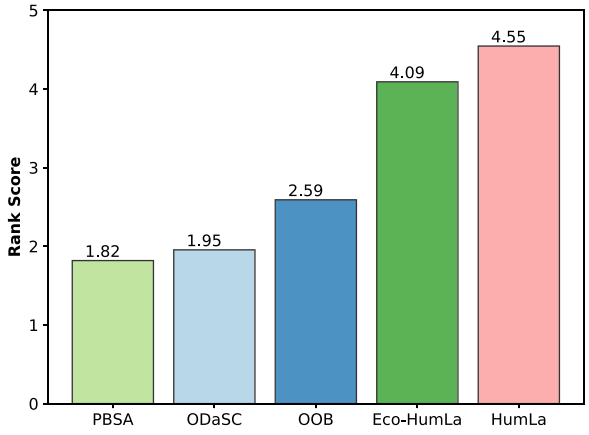
A central finding of this study is that BDFE yields substantially more accurate feature values, yet this improved accuracy produces only small-effect differences in predictive performance (consistently small A_{12} effect sizes) and leaves model rankings highly consistent (median Kendall's $\tau > 0.8$). We stress that this is a *finding*, not a limitation of BDFE: Because BDFE supplies accurate, branch-aware feature values, we are able to establish,



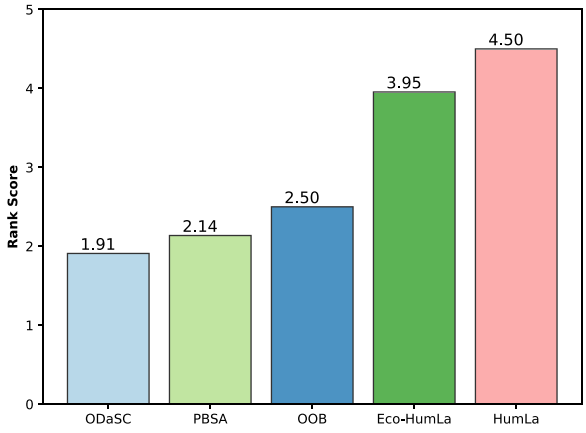
(a) G-Mean Rank Score on noisy feature values F^-



(b) G-Mean Rank Score on new feature values F



(c) MCC Rank Score on noisy feature values F^-



(d) MCC Rank Score on new feature values F

FIGURE 7 | Rank score for five models.

for the first time, that the noise introduced by chronological extraction does not materially distort model comparison. By the same token, the consistently small effect sizes are reassuring for the community. They provide positive evidence that prior empirical JIT-SDP studies built on conventionally extracted datasets are not invalidated by branch-dependency noise; their conclusions remain robust and do not require costly re-validation. The value of BDFE therefore lies in *measurement validity* rather than in raw predictive gain. This naturally raises a practical question: *Given that the two extraction methods yield comparable predictive accuracy, when should researchers or practitioners prefer the more accurate but more costly BDFE?* We offer guidance along the following dimensions.

Explainability and edge-case debugging. When a deployed defect-prediction system must explain why a particular change was predicted, the explanation typically refers to specific feature values (e.g., a change that modified a file with a very high NUC value). If those values are miscomputed because branch dependency was ignored, the explanation can be wrong and developer trust is undermined. By providing more *trustworthy* per-commit feature values, BDFE supports better local explanations, which matters a lot in high-stakes projects where suspect changes are reviewed manually.

Construction of long-term benchmark datasets. Building a definitive, high-quality benchmark dataset for JIT-SDP requires accurate feature extraction. Although chronological extraction is acceptable for comparing models, as our ranking-consistency results demonstrate, a near-to gold-standard dataset should be free from avoidable noise. BDFE provides a more reliable foundation for such curated datasets, ensuring that future studies start from the most accurate feature values available.

Model robustness analysis. The five models evaluated here are ensemble methods built on decision trees, which average predictions over many bootstrap samples and split on local thresholds; both properties make them robust to small feature perturbations, as we analyze in Section 6.2.2. Models that lack this averaging or partitioning behavior, such as linear models or gradient-based deep networks that consume raw feature magnitudes, may be more sensitive to feature-level noise. Constructing

datasets with BDFE from the outset removes one potential source of confounding and makes future comparisons across such model families fairer.

Cost–benefit consideration. As reported in Table 5. BDFE increases extraction time by a median of about 87% (range + 20% to + 349%, i.e., roughly 1.2× to 4.5×) while leaving memory consumption essentially unchanged (below 3% for 21 of the 22 projects). In absolute terms this adds at most about two minutes per 1000 commits, a one-time cost incurred during offline dataset construction rather than at prediction time. Because the time complexity remains linear ($O(N)$) with only a bounded constant factor that depends on repository branching, the overhead scales predictably and is negligible relative to downstream training and evaluation. For any setting in which data quality is prioritized over raw extraction speed, the cost of BDFE is therefore well justified.

Summary. We do not claim that BDFE should always replace chronological extraction. Chronological extraction remains a reasonable choice for rapid prototyping, or when the objective is to reproduce or compare existing model rankings, for which our results show the two methods agree. BDFE is the preferable choice whenever feature values are *used directly*: (1) in feature-importance or correlation analysis, (2) in local explanations, (3) in the construction of gold-standard benchmark datasets, or 4) when evaluating models that may be more sensitive to feature-level noise.

7.2 | Continued Relevance Under Semantic and LLM-Based Features

JIT-SDP increasingly employs semantic features derived from deep learning and LLMs [38, 59, 60]. This raises two questions regarding the scope of our work: whether the branch-dependency correction remains relevant when code changes are represented as *embeddings* rather than expert-crafted metrics, and whether the rise of LLM-based approaches risks making such feature engineering obsolete.

We contend that BDFE remains directly *relevant*—and may matter even more—for semantic feature extraction. Many SOTA embedding-based approaches (e.g., CC2Vec [31], JITLine [32], CCT5 [37]) operate on representations of code changes, and therefore need the precise pre- and post-commit code to compute deltas or contextual embeddings. Correct pre-change code depends on identifying the true parent commit in the revision graph; chronological misidentification distorts the entire input to the embedding model. For expert features such an error perturbs only a few numeric values—which, as our results show, has limited downstream impact (Section 7.1); whereas, for semantic pipelines it corrupts the whole representation, so the effect could be considerably larger. The trend toward fine-grained prediction at the file or hunk level [51] likewise relies on accurate change boundaries, which are inherently tied to branch dependency. BDFE therefore offers a foundational preprocessing step that benefits both expert- and semantic-feature-based pipelines.

Nor does the growing emphasis on LLMs render expert features obsolete. Recent hybrid approaches combine expert features with semantic embeddings to boost performance [33, 34], and in such multimodal model settings, ensuring the expert features are noise-free, which is precisely what BDFE provides, can improve overall robustness. We therefore view branch-aware extraction not as a competitor to semantic methods, but as a *complementary* data-quality layer that stays valuable as the field moves toward LLM-based prediction.

8 | Threats To Validity

Internal validity: Each prediction method was executed 30 times for each dataset to mitigate threats to internal validity arising from the randomness inherent in the model construction. Another potential threat concerns the accuracy of defect-inducing labels, as some defects may never be confirmed, particularly when the defects they cause are not induced until the end of the data stream due to very large verification latency. To mitigate this threat, we selected open-source projects spanning a long period of four years and excluded software changes from the last 6 months of the data streams.

External validity: We have investigated 22 projects using five SOTA JIT-SDP models, covering a wide range of software product characteristics and advanced prediction techniques. As with any study involving machine learning, the results may not generalize well to other types of projects or prediction models. Future research could explore additional JIT-SDP methods that operate in online scenarios using the same investigation procedures. Similarly, replicating the analysis on a broader set of open-source projects would help further validate our findings.

Construct validity: We use G-Mean to evaluate predictive performance of JIT-SDP, as it is an unbiased metrics well-suited for the class imbalance problem inherent in JIT-SDP. Additionally, we have also adopted MCC, a highly recommended metric in the SDP literature, which consider all entries of the confusion matrix. Other performance metrics, including Recall0, Recall1, precision, and F1-score, were also analyzed in our study and yielded consistent results. Due to space limitation, these metrics have been omitted in the main paper, and are presented in the supplementary material. A fading factor is adopted to enable tracking the fluctuations in predictive performance over time, as recommended for online learning [52].

9 | Conclusion

This paper systematically investigates a specific type of feature noise arising from the negalect of branch dependency, an issue frequently overlooked by most feature extraction methods in the JIT-SDP community. We conducted an extensive experimental study to explore the impact of this feature noise on the predictive performance of JIT-SDP by answering three RQs. Our key findings and their implications are summarized as follows.

RQ1. To what extent does neglecting branch dependency during feature extraction affect the accuracy of feature

values? Our findings indicate that five features, namely, LT, NDEV, AGE, NUC, and REXP, are significantly impacted by this issue. Although the effect sizes for these features are small in terms of A12 (all being less than 0.56), we can not ignore that. If researchers wish to obtain more precise feature data, or if their researches are data-sensitive, we recommend they consider branch dependencies during the feature extraction process to achieve more accurate features. Besides considering branch dependency, we believe that the quality of features can still be further enhanced, and we encourage researchers to conduct more studies on this matter.

RQ2. Given the potential significant impact on extracted feature values, how do noisy features affect the predictive performance of JIT-SDP models? Feature noise arising from neglecting branch dependency significantly affects predictive performance of JIT-SDP models except for Eco-HumLa. Our conclusions show that the feature noise does not have a serious impact on the actual performance of the model, but it is still significant for some models. The 14 features are themselves generalized features of the software commit attributes, which are high-level description of the expert features, although five of the features are affected to some extent, they do not have an absolute impact on the overall performance. On the one hand, the results of the permutation importance showed that the overall effort of these five affected features to SDP prediction is less than the sum of the other features; On the other hand, it also demonstrates the high robustness of ensemble models and tree models in this noisy environment, which is a positive result for researchers, as it means that the performance of previously related models has not been seriously affected. However, we still suggest that researchers should not completely ignore the impacts on these five sets of features, as some models may focus more on specific values of features than on the overall distribution, or may be more affected by these five features.

RQ3. If feature noise significantly impacts performance, what implications does this have for the conclusions regarding the relative rankings of various JIT-SDP models? Feature noise resulting from the absence of branch dependency does not have significant impact on the overall rankings of JIT-SDP models, indicating that such feature noise is unlikely to affect the validity and accuracy of previous academic studies, especially the comparison of the advantages and disadvantages of the performance. Although the current conclusion suggests that experimental results on the pros and cons of the model will not be affected, we still recommend that researchers use BDFE and the new features.

Moreover, to explore these three RQs, we proposed a novel feature extraction method called BDFE, which is specifically designed to incorporate branch dependency into the feature extraction process in Section 4. Our results showed that the BDFE method produces significantly more accurate feature values than those generated by the widely used tool `Commit Guru`. This improved feature extraction method has been made publicly available to support future research in JIT-SDP.

We also provided a thorough justification for using our BDFE features as ground-truth feature values when addressing those RQs. Our BDFE features are more precise than those from `Commit Guru`, and given the sheer volume of commits and challenges associated with determining some feature values under specific circumstances (as discussed in Section 5.3), it is impractical to manually inspect all commits to obtain accurate ground-truth feature values.

Future work will involve conducting empirical studies on alternative data extraction tools, such as the one proposed by Borg et al. [29]; investigating additional factors that may affect feature quality in JIT-SDP, including the impact of merging commits and the use of **git blame**; and incorporating semantic features of code changes through deep or LLMs [34, 59].

Author Contributions

Zuowei Chen: methodology design and implementation, experimental execution, data analysis, and manuscript drafting, contributed to rebuttal preparation under supervision. **Liyan Song:** conceptualization and supervision of the research workflow including methodology, experimental design and analysis, contributed to major manuscript revision and lead the response to reviewers.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (NSFC) under Grant Nos. 62572154 and 62002148, the Heilongjiang Province Natural Science Foundation under Grant No. LH2024F016, and the Harbin Institute of Technology Talent Start-up Project under Grant No. AUGA5710010924.

Data Availability Statement

The data that support the findings of this study are available from the corresponding author upon reasonable request.

References

1. S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying Software Changes: Clean or Buggy?," *IEEE Transactions on Software Engineering* 34, no. 2 (2008): 181–196.

2. P. L. Li, J. Herbsleb, M. Shaw, and B. Robinson, "Experiences and Results From Initiating Field Defect Prediction and Product Test Prioritization Efforts at ABB Inc.," in *Proceedings of the 28th International Conference on Software Engineering* (ACM, 2006), 413–422.

3. A. E. Hassan, "Predicting Faults Using the Complexity of Code Changes," in *2009 IEEE 31st International Conference on Software Engineering* (IEEE, 2009), 78–88.

4. Y. Kamei, E. Shihab, B. Adams, et al., "A Large-Scale Empirical Study of Just-in-Time Quality Assurance," *IEEE Transactions on Software Engineering* 39, no. 6 (2013): 757–773.

5. Y. Zhao, K. Damevski, and H. Chen, "A Systematic Survey of Just-in-Time Software Defect Prediction," *ACM Computing Surveys* 55, no. 10 (2023): 1.1–1.35.

6. E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, *An Industrial Study on the Risk of Software Changes*, vol. 1 (ACM, 2012).

7. Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin, and X. Yang, *Perceptions, Expectations, and Challenges in Defect Prediction* (IEEE Transactions on Software Engineering, 2018), 1–1.

8. A. Mockus and D. M. Weiss, "Predicting Risk of Software Changes," *Bell Labs Technical Journal* 5, no. 2 (2000): 169–180.

9. M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online Defect Prediction for Imbalanced Data," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2 (IEEE, 2015), 99–108.

10. Y. Kamei and E. Shihab, "Defect Prediction: Accomplishments and Future Challenges," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5 (IEEE, 2016), 33–45.

11. G. G. Cabral, L. L. Minku, E. Shihab, and S. Mujahid, "Class Imbalance Evolution and Verification Latency in Just-in-Time Software Defect Prediction," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (IEEE, 2019), 666–676.

12. G. G. Cabral and L. L. Minku, "Towards Reliable Online Just-in-Time Software Defect Prediction," *IEEE Transactions on Software Engineering* 49, no. 3 (2022): 1342–1358.

13. S. McIntosh and Y. Kamei, "Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-in-Time Defect Prediction," *IEEE Transactions on Software Engineering* 44, no. 5 (2018): 412–428.

14. C. Rosen, B. Grawi, and E. Shihab, "Commit Guru: Analytics and Risk Prediction of Software Commits," in *Joint Meeting on Foundations of Software Engineering* (ACM, 2015), 966–969.

15. L. Song and L. L. Minku, "A Procedure to Continuously Evaluate Predictive Performance of Just-in-Time Software Defect Prediction Models During Software Development," *IEEE Transactions on Software Engineering* 49, no. 2 (2023): 646–666.

16. L. Song, S. Li, L. L. Minku, and X. Yao, "A Novel Data Stream Learning Approach to Tackle One-Sided Label Noise From Verification Latency," in *2022 International Joint Conference on Neural Networks (IJCNN)* (IEEE, 2022), 1–8.

17. G. G. Cabral, L. L. Minku, A. L. Oliveira, D. A. Pessoa, and S. Tabassum, "An Investigation of Online and Offline Learning Models for Online Just-in-Time Software Defect Prediction," *Empirical Software Engineering* 28, no. 5 (2023): 121.

18. L. Song, L. L. Minku, C. Teng, and X. Yao, "A Practical Human Labeling Method for Online Just-in-Time Software Defect Prediction," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ACM, 2023), 605–617.

19. S. Tabassum, L. L. Minku, and D. Feng, "Cross-Project Online Just-in-Time Software Defect Prediction," *IEEE Transactions on Software Engineering* 49, no. 1 (2022): 268–287.

20. Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying Just-in-Time Defect Prediction Using Cross-Project Models," *Empirical Software Engineering* 21 (2016): 2072–2106.

21. S. Wang, L. L. Minku, and X. Yao, "Resampling-Based Ensemble Methods for Online Class Imbalance Learning," *IEEE Transactions on Knowledge and Data Engineering* 27, no. 5 (2015): 1356–1368.

22. S. Tabassum, L. L. Minku, D. Feng, G. G. Cabral, and L. Song, "An Investigation of Cross-Project Learning in Online Just-in-Time Software Defect Prediction," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (ACM, 2020), 554–565.

23. T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi, "An Empirical Study of Just-in-Time Defect Prediction Using Cross-Project Models," in *Proceedings of the 11th Working Conference on Mining Software Repositories* (ACM, 2014), 172–181.

24. A. T. Misirli, E. Shihab, and Y. Kamei, "Studying High Impact Fix-Inducing Changes," *Empirical Software Engineering* 21 (2016): 605–641.

25. J. Sliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?," *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005* 30, no. 4 (2005): 1–5.

26. S. Kim, T. Zimmermann, K. Pan, and E. J. W. Jr, "Automatic Identification of Bug-Introducing Changes," in *Proc International Conference on Automated Software Engineering* (IEEE, 2006).

27. E. C. Neto, D. A. D. Costa, and U. Kulesza, "The Impact of Refactoring Changes on the SZZ Algorithm: An Empirical Study," in *IEEE International Conference on Software Analysis* (IEEE, 2018).

28. D. A. D. Costa, S. Mcintosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes," *IEEE Transactions on Software Engineering* 43, no. 7 (2017): 641–657.

29. M. Borg, O. Svensson, K. Berg, and D. Hansson, "SZZ Unleashed: An Open Implementation of the SZZ Algorithm—Featuring Example Usage in a Study of Just-in-Time Bug Prediction for the Jenkins Project," in *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation* (ACM, 2019), 7–12.

30. S. Herbold, A. Trautsch, F. Trautsch, and B. Ledel, "Problems With SZZ and Features: An Empirical Study of the State of Practice of Defect Prediction Data Collection," *Empirical Software Engineering* 27, no. 2 (2022): 42.

31. T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed Representations of Code Changes," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (ACM, 2020), 518–529.

32. C. Pornprasit and C. K. Tantithamthavorn, "Jitline: A Simpler, Better, Faster, Finer-Grained Just-in-Time Defect Prediction," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)* (IEEE, 2021), 369–379.

33. X. Zhou, D. Han, and D. Lo, "Bridging Expert Knowledge With Deep Learning Techniques for Just-in-Time Defect Prediction," *Empirical Software Engineering* 30 (2025): 37.

34. 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 *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ACM, 2022), 672–683.

35. A. M'baya and N. Moalla, "Deep Semantic and Strutural Features Learning Based on Graph for Just-in-Time Defect Prediction," in *ENASE: Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering*, eds. H. Kaindl, M. Mannion, and L. Maciaszek (SCITEPRESS, 2022), 128–137.

36. X. Chen, H. Xia, W. Pei, C. Ni, and K. Liu, "Boosting Multi-Objective Just-in-Time Software Defect Prediction by Fusing Expert Metrics and Semantic Metrics," *Journal of Systems and Software* 206 (2023): 111853.

37. B. Lin, S. Wang, Z. Liu, Y. Liu, X. Xia, and X. Mao, "Cct5: A Code-Change-Oriented Pre-Trained Model," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ACM, 2023), 1509–1521.

38. L. Fan, J. Liu, Z. Liu, D. Lo, X. Xia, and S. Li, "Exploring the Capabilities of Llms for Code-Change-Related Tasks," *ACM Transactions on Software Engineering and Methodology* 34, no. 6 (2025): 1–36.

39. S. Kim and E. J. Whitehead, Jr., "How Long Did It Take to Fix Bugs?" in *Proceedings of the 2006 International Workshop on Mining Software Repositories* (ACM, 2006), 173–174.

40. J. Eyolfson, L. Tan, and P. Lam, "Do Time of Day and Developer Experience Affect Commit Bugginess?" in *Proceedings of the 8th Working Conference on Mining Software Repositories* (ACM, 2011), 153–162.

41. P. Sedgwick, "Pearson's Correlation Coefficient," *BMJ* 345 (2012): e4483–e4483.

42. F. Wilcoxon, "Individual Comparisons by Ranking Methods," in *Breakthroughs in Statistics: Methodology and Distribution* (Springer, 1992), 196–202.

43. Y. Benjamini and D. Yekutieli, "The Control of the False Discovery Rate in Multiple Testing Under Dependency," *Annals of Statistics* 29, no. 4 (2001): 417–442.

44. A. Vargha and H. D. Delaney, "A Critique and Improvement of the *CL* Common Language Effect Size Statistics of Mcgraw and Wong," *Journal of Educational and Behavioral Statistics* 25, no. 2 (2000): 101–132.

45. M. Kubat, R. Holte, and S. Matwin, "Learning When Negative Examples Abound," in *Machine Learning: ECML-97: 9th European Conference on Machine Learning Prague*, vol. 9 (Springer, 1997), 146–153.

46. S. Wang, L. L. Minku, and X. Yao, "A Systematic Study of Online Class Imbalance Learning With Concept Drift," *IEEE Transactions on Neural Networks and Learning Systems* 29, no. 10 (2018): 4802–4821.

47. Q. Zhu, "On the Performance of Matthews Correlation Coefficient (MCC) for Imbalanced Dataset," *Pattern Recognition Letters* 136 (2020): 71–80.

48. D. Chicco, M. Warrens, and G. Jurman, "The Matthews Correlation Coefficient (MCC) Is More Informative Than Cohen's Kappa and Brier Score in Binary Classification Assessment," *IEEE Access* 9 (2021): 78368–78381.

49. M. Kondo, D. M. German, O. Mizuno, and E.-H. Choi, "The Impact of Context Metrics on Just-in-Time Defect Prediction," *Empirical Software Engineering* 25 (2020): 890–939.

50. M. Kondo, Y. Kashiwa, Y. Kamei, and O. Mizuno, "An Empirical Study of Issue-Link Algorithms: Which Issue-Link Algorithms Should We Use?," *Empirical Software Engineering* 27, no. 6 (2022): 136.

51. X. Zhu, C. Yan, E. J. Whitehead, Jr., B. Niu, L. Zhu, and L. Pan, "Just-in-Time Defect Prediction for Software Hunks," *Software: Practice and Experience* 52, no. 1 (2022): 130–153.

52. J. Gama, R. Sebastiao, and P. P. Rodrigues, "On Evaluating Stream Learning Algorithms," *Machine Learning* 90 (2013): 317–346.

53. M. G. Kendall, "A New Measure of Rank Correlation," *Biometrika* 30, no. 1–2 (1938): 81–93.

54. A. Altmann, L. Toloşi, O. Sander, and T. Lengauer, "Permutation Importance: A Corrected Feature Importance Measure," *Bioinformatics* 26, no. 10 (2010): 1340–1347.

55. S. Guo, D. Li, L. Huang, et al., "Estimating Uncertainty in Labeled Changes by SZZ Tools on Just-in-Time Defect Prediction," *ACM Transactions on Software Engineering and Methodology* 33, no. 4 (2024): 1–25.

56. N. Oza, "Online bagging and boosting," in *2005 IEEE International Conference on Systems, Man and Cybernetics*, vol. 3 (IEEE, 2005), 2340–2345.

57. J. R. Quinlan, "Induction of Decision Trees," *Machine Learning* 1, no. 1 (1986): 81–106.

58. L. Breiman, "Bagging Predictors," *Machine Learning* 24, no. 2 (1996): 123–140.

59. S. Liu, J. Keung, Z. Yang, F. Liu, Q. Zhou, and Y. Liao, "Delving Into Parameter-Efficient Fine-Tuning in Code Change Learning: An Empirical Study," arXiv preprint arXiv:2402.06247, 2024.

60. S. Kwon, J.-I. Jang, S. Lee, D. Ryu, and J. Baik, "Codebert Based Software Defect Prediction for Edge-Cloud Systems," in *Current Trends in Web Engineering, ICWE 2022 International Workshops 1668 of Communications in Computer and Information Science 2023. 22nd International Conference on Web Engineering (ICWE)*, eds. G. Agapito, A. Bernasconi, C. Cappiello, et al. (Springer, 2022), 11–21.

Supporting Information

Additional supporting information can be found online in the Supporting Information section. supplement.tex supplement.pdf