Leveraging Commit-Size Context and Hyper Co-Change Graph Centralities for Defect Prediction

***Abstract*—File-level defect prediction models traditionally rely on product and process metrics. While process metrics effectively complement product metrics, they often overlook *commit size*—the number of files changed per commit—despite its strong association with software quality. Network centrality measures on dependency graphs have also proven to be valuable product-level indicators. Motivated by this, we first redefine process metrics as *commit-size–aware process metric vectors*, transforming conventional scalar measures into 100-dimensional profiles that capture the distribution of changes across commit-size strata. We then model change history as a *hyper co-change graph*, where hyperedges naturally encode commit-size semantics. Vector centralities computed on these hypergraphs quantify size-aware node importance for source files. Experiments on nine long-lived Apache projects (32 releases) using five popular classifiers show that replacing scalar process metrics with the proposed commit-size–aware vectors, alongside product metrics, consistently improves predictive performance. Median gains range from 3.5–7.3% in AUROC, 5.6–26.7% in AUPRC, 8.0–17.6% in F1, and 11.5–28.3% in MCC, while Brier scores decrease by 3.0–11.3%. Incorporating hyper co-change graph centralities with product and vector process metrics achieves the best overall results—yielding the highest AUROC, MCC, and F1 in 75.6%, 66.7%, and 71.1% of cases, respectively. Friedman and post-hoc Nemenyi tests confirm these improvements are statistically significant ($p < 0.05$). These findings establish that commit-size–aware process metrics and hypergraph-based vector centralities capture higher-order change semantics, leading to more discriminative, better-calibrated, and statistically superior defect prediction models.**

***Index Terms*—Software defect prediction, process metrics, commit-size–aware process metrics, hyper co-change graph, vector centrality.**

I. Introduction

Software Defect prediction models primarily use two types of metrics: product and process metrics. Product metrics, derived from code analysis, characterize the structural properties of software using measures such as McCabe's cyclomatic complexity [1] and the CK suite of object-oriented metrics [2], among numerous metrics proposed to evaluate software quality. In contrast, process metrics, such as commit count and change entropy [3], [4], are derived from the project's historical change data and capture the temporal dynamics of software evolution, reflecting the intensity, scope, and distribution of code modifications. Numerous studies have shown that process metrics significantly enhance the performance of defect prediction and effectively complement product metrics [4]–[6]. Moreover, they are language-independent and adaptable across diverse projects, making them an indispensable component of defect prediction models.

Although conventional process metrics for file-level defect classification capture important aspects of software change behavior—such as the number of commits and the number of developers who modified a file—they generally overlook an important factor: the size of the commits (i.e., the number of source code files changed in each commit) that affected the file, which is closely associated with software quality. Prior research has demonstrated that commit size, measured by the number of files changed, influences design quality [7], affects the success of automated software repair [8], and is correlated with bug reopens, as bugs fixed in larger commits tend to be reopened more often [9]. Larger commits have also been identified as indicators of code decay, architectural degradation, and erosion of software design quality [10]–[12], and are more bug-inducing than smaller ones [13].

However, while the number of files changed in a commit (NF) has been a widely used metric in *Just-In-Time (JIT) defect prediction*—which predicts whether a commit made or proposed by a developer is defect-prone [14]–[17]—it is rarely incorporated into file-level defect prediction. Existing process metrics typically rely on coarse-grained measures, such as the number of commits or the total lines added and deleted in the previous release. However, commits of different sizes can introduce defects with varying likelihoods and impact [18], and aggregating such change information into simple count-based metrics obscures these differences.

Figure 1 illustrates this with an example where both *InheritedCacheCompiler.java* and *ChannelStream.java* in release 1.4.0 of the JRuby project (analyzed in this study) were modified 40 times, yielding identical commit counts. However, *ChannelStream.java* was changed predominantly in smaller commits, whereas *InheritedCacheCompiler.java* was affected by several large commits. A commit-size–agnostic metric such as commit count treats these files as equivalent, overlooking the underlying distribution of commit sizes that may influence their differing defect proneness.

This observation motivates the incorporation of commit-size information into process metrics to improve file-level defect prediction. To this end, we extend conventional process metrics to be *commit-size–stratified*, thereby enhancing their explanatory and predictive capabilities. Specifically, we transform widely used scalar process metrics [4], [6] into *commit-size–stratified change profiles* represented as vectors. For each file, every component of the vector corresponds to the value of the metric computed exclusively from commits within a specific commit-size bin. This stratification preserves the distributional characteristics of the metric across commits

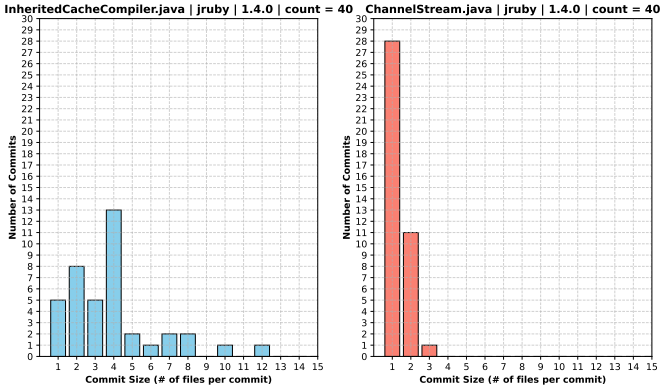


Fig. 1: Commit size distribution for two files with identical commit counts in JRuby 1.4.0.

of different magnitudes—contextual information that a single scalar aggregation inherently discards.

The contribution of this study is an empirical evaluation of whether these commit-size–aware vector process metrics, when integrated with product metrics, can enhance the discriminatory power and predictive performance of file-level defect prediction models compared to their scalar counterparts. Accordingly, we formulate our first research question as follows:

RQ1: *Do commit size–aware vector process metrics complement product metrics more effectively than their scalar counterparts in improving defect prediction performance?*

Software entities that frequently change together are said to exhibit a *change dependency* [19]. The impact of such co-change relationships on software quality and defect occurrence has been extensively studied. These relationships, where files are modified together within the same commit, are typically modeled as *co-change graphs*. Prior studies have shown that analyzing the structural and topological properties of these graphs aids in understanding software evolution and predicting future defects [20]–[24].

Some of the most popular measures or metrics in graph theory come from Social network anlaysis (SNA) which are node centrality measures like degree centrality, betweenness centrality, etc., to measure the node importance in the network. They measure the importance of the node in the network. The node centrality measures on structural/code dependency networks have been used extensively as metrics for defect prediction in the source code files [25]–[28]. However, although effective on structural dependency networks, the utility of these centrality measures for defect prediction on co-change networks remains under-evaluated.

Simple co-change (pairwise) networks ignore commit size, so they cannot distinguish a file co-changed in numerous small commits from one co-changed in a few large commits. In a hypernetwork [29][1], the number of nodes participating in an edge is not limited to two, extending the pairwise model. Representing each commit as a hyperedge linking all modified files captures higher-order group interactions and inherently encodes commit-size semantics.

Among the various node importance measures proposed for hypernetworks, *vector centrality* [30] is particularly well suited for modeling co-change behavior and capturing commit-size semantics. Unlike traditional scalar measures, it represents each node's importance as a vector across hyperedges of different sizes, reflecting its diverse roles in development and maintenance. Higher values in lower dimensions indicate frequent participation in small commits involving few files, whereas higher-dimensional values signify involvement in large, system-wide, or *cross-cutting* modifications that span multiple modules. Consequently, vector centrality captures a file's participation across diverse commit contexts and, in doing so, naturally integrates commit size semantics for a richer change-history profile.

Motivated by the ability of hypernetworks to capture higher-order interactions and the potential of vector centrality to integrate commit-size semantics into node importance, we pose our second research question as follows:

RQ2: *What impact does incorporating vector centralities derived from hyper-co-change networks have on the performance of defect prediction models built upon product and vector process metrics?*

In summary, this paper makes the following key contributions:

- **Commit size–aware vectorization of process metrics:** To the best of our knowledge, we are the first to incorporate the *size of the commit* into the computation of process metrics by redefining traditional scalar metrics as *commit size–aware vectors*. This formulation captures file-level change behavior across varying commit magnitudes, preserving richer process semantics. Empirical evaluation across nine long-lived projects (32 releases) shows that the proposed vector process metrics complement product metrics more effectively than their scalar counterparts, leading to significant improvements in defect prediction.
- **Hyper–co-change network centralities for enhanced prediction:** To the best of our knowledge, we are the first to model co-change activities as a *hypergraph* and employ *vector centralities* (degree, betweenness, closeness, eigenvector) to compute commit size–aware node importance measures for defect prediction. These hypergraph-based vector centralities capture higher-order co-change relationships among files modified together in commits. Incorporating them alongside product and vector process metrics yields the best overall prediction performance, with statistically significant gains validated through Friedman and post-hoc Nemenyi analyses.

II. Related Work

There are two prominent categories of metrics used in software defect prediction: *product* and *process* metrics. Product

[1]"Hypernetworks" and "hypergraphs" are used synonymously in this paper

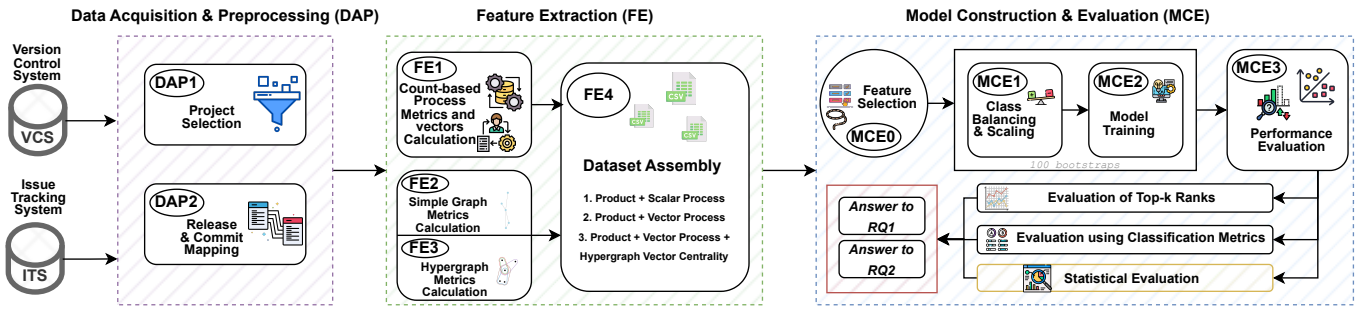


Fig. 2: Schematic representation of the overall methodological framework

metrics, derived from static code analysis, characterize the structural properties of software, whereas process metrics, such as commit count, are obtained from a project's historical change data. Prior research has shown that process metrics often outperform product metrics in predictive accuracy. Rahman and Devanbu [4] provided early evidence of this trend, later reinforced by large-scale studies such as Majumder et al. [6]. Moreover, Majumder et al. [31] demonstrated that process metrics remain effective even under semi-supervised settings with limited labeled data.

Changes made to the system form the foundation of process metrics. Hassan [3] showed that systems with more distributed changes across files are more defect-prone, suggesting that file-level change information is a valuable defect indicator. Nagappan et al. [32] found that change bursts—frequent changes within short time intervals—are strong predictors of defects. Similarly, Wen et al. [33] employed recurrent neural networks to model sequential change patterns and demonstrated that both the amount and the temporal sequence of changes influence defect likelihood.

Several studies have also highlighted that the number of files co-changed with a given file (i.e., commit size) is indicative of post-release defects and software quality. Moser et al. [34] incorporated the average and maximum number of co-changed files as features and found them useful for defect prediction. Kouroshfar et al. [35] observed that the number of co-changed files can serve as an indicator of architectural quality. Shihab et al. [36] noted that high-impact bugs tend to involve larger commits, while Yan et al. [37] proposed using the number of files modified to estimate defect-fixing effort in effort-aware defect prediction. However, to the best of our knowledge, no prior work has comprehensively modeled the *distribution* of commit sizes a file has experienced or utilized its commit-size-aware change history for defect prediction. Our study fills this gap by constructing a commit-size–aware change profile for each file and leveraging it for defect prediction.

More closely related to our work is by Šikić et al. [38] who considered aggregated change metrics based on the sizes of commits involving each file. However, they summarized commit sizes using average values, overlooking the fine-grained distribution of small versus large commits, which can be crucial for understanding a file's defect-proneness. In contrast, our approach preserves the more detailed distributional information, enabling a richer representation of a file's evolutionary behavior.

Commit size has also been extensively studied in *Just-In-Time (JIT) defect prediction* [14]–[17]. However, since our focus is on file-level defect prediction, we only briefly refer to this line of work in this section without discussing it in detail.

Applying network analysis and centrality measures for defect prediction at the product level or within developer communication networks is not new. Network centrality measures on code dependency graphs and developer networks have been extensively applied in prior research [25], [26], [28], [39], [40]. However, since our work focuses primarily on process metrics, we do not discuss these studies in detail.

Simple co-change networks and analyses on them have been used in prior studies to investigate software quality [21], [23], [41], [42]. However, simple pairwise networks cannot accommodate commit size, and this aspect is therefore lost. Our work extends this research direction by leveraging *hyper co-change networks* that capture higher-order relationships among files modified together in a commit. Using this hypergraph representation, we compute *vector centrality* [30] for each file, capturing its role across commits of different sizes and thus encoding both structural and semantic aspects of changes.

Finally, hypergraph-based models have recently gained attention in software engineering [43]–[47]. For instance, Qiu et al. [48] used a hypergraph neural network for defect prediction, though their model mainly focuses on code-level (product) features rather than process data. In contrast, we model co-changes as a hypergraph and employ vector centrality to measure node importance. To the best of our knowledge, this is the first study to apply hypergraph-based vector centrality on process data for defect prediction.

In summary, our research advances prior work by incorporating commit context—specifically, the size of commits in which files were changed—through vector process metrics and by modeling higher-order co-change relationships using a hyper-co-change graph for defect prediction.

III. Methodology

In this section, we outline the methodology used to address our research questions. We first describe the dataset and

preprocessing steps, followed by the classification models and validation techniques. Finally, we detail the experimental setup employed to answer our research questions. The overall workflow of the study is illustrated in Figure 2, which provides a schematic overview of the complete methodological framework.

A. Dataset Description and Preprocessing

We used the dataset prepared by Yatish et al. [49] for our experiments. This dataset was chosen because the authors carefully selected projects and releases where most issue reports are either closed or fixed and linked to their corresponding code changes, ensuring high data quality. Additionally, it provides 54 product metrics essential for our analysis. Since our objective is to examine whether incorporating commit-size information into process metrics and size-aware node centrality measures complements existing product metrics to enhance defect prediction performance, this dataset is ideally suited for our study. It is also widely used in prior defect prediction research. [28], [50]–[55]. The details of our dataset is shown in table I. We primarily used this dataset to obtain product metrics for source code files and their post-release defect labels. However, all process metrics were computed separately from the commit data after cloning the projects.

To compute the conventional and vector process metrics, as well as the hypergraph-based vector centralities of source code files derived from co-change information, we first analyze the version control data—specifically, the commits and their associated metadata. However, in defect prediction studies, a recurring challenge lies in deciding which commits should be included or excluded when computing features and constructing the training dataset for predicting future defects. Selecting an appropriate threshold is critical: very large commits are often associated with administrative activities such as repository restructuring or dependency refactoring [56]. Including such commits can inflate noise and lead to false positives by incorrectly labeling defect-free files as defective, whereas using too restrictive a threshold may exclude relevant commits and overlook genuinely defective files [57]. Consequently, several recent studies have adopted a standard threshold of 100 files per commit, excluding commits that modify more than 100 files from defect prediction datasets [56], [58]–[61]. Following this convention, we also excluded all commits that changed more than 100 files from our final dataset.

B. Classification and Model Validation

We employ five widely used classifiers in our study—Logistic Regression, Support Vector Machine (SVM), XGBoost, Gradient Boosting Machine (GBM), and Random Forest. Logistic Regression serves as a strong, interpretable linear baseline with built-in regularization and well-calibrated probabilities. SVM is a margin-based learner that performs effectively in high-dimensional, sparse feature spaces common to software metrics. XGBoost, a regularized gradient-boosted tree model, captures complex feature interactions and handles missing values efficiently. GBM offers a flexible boosting framework that balances bias and variance, yielding robust and calibrated predictions. Random Forest, an ensemble of bagged decision trees, mitigates overfitting and provides stable feature importance estimates under noisy metric sets. Collectively, these models span linear and ensemble-based paradigms and have been extensively validated in prior defect prediction studies [62]–[65]. We used the `Scikit-learn` library [66] implementation for all the classifiers in our study.

To ensure robustness and minimize bias in our results, we employ an out-of-sample bootstrap validation technique. This method has been widely recommended for producing stable and unbiased estimates in defect prediction studies [28], [49], [62], [67], [68]. For a dataset with D instances, each bootstrap sample is generated by randomly selecting D instances with replacement, leaving out roughly 36% of the instances as unseen (out-of-bag) test data. The model is trained and evaluated on 100 such bootstrap samples, and the final performance is reported as the average across these iterations.

C. Handling Data Imbalance and Feature Selection

Before training classifiers and evaluating their performance, we perform two key preprocessing steps. First, since defect datasets often suffer from class imbalance—where buggy files are significantly fewer than non-buggy ones—we apply the Synthetic Minority Over-sampling Technique (SMOTE) [69] to balance the data. SMOTE is widely and successfully used in defect prediction studies [6], [70].

To reduce noise and redundancy among features—which are often interdependent—we apply feature selection before training the classifiers. Several feature selection techniques have been explored for defect prediction [71]. Xu et al. [71] reported that Correlation-based Feature Selection (CFS) performs well in many cases, though its effectiveness varies across datasets. Jiarpakdee et al. [72] proposed *AutoSpearman* as an effective feature selection method for defect prediction. However, since correlation-based feature selection methods and AutoSpearman rely on pairwise correlations, they do not scale efficiently to high-dimensional settings such as ours, which involves over 1,800 features. Therefore, we adopt the Hilbert–Schmidt Independence Criterion (HSIC) Lasso [73], a kernel-based method well suited for high-dimensional data that can capture nonlinear dependencies between features and target variables and is widely used for feature selection in machine learning.

D. Experimental Setup to Answer RQ1

1) Details of the Metrics: RQ1 investigates whether size-aware vector process metrics provide improvements in defect classification over their scalar counterparts and whether they complement popular product metrics more effectively. To address this, we first extracted the commits corresponding to each release of every project in our dataset. The projects were cloned from their respective GitHub repositories, and commits were retrieved using tag information and the `git rev-list`

TABLE I: Dataset Description

Project	Description	Commits	Src Files	Bugs	Defective Rate	Releases
Activemq	Messaging and Integration Patterns server	174-634	88-1847	1626	15.54%-67.05%	5.0.0, 5.1.0, 5.2.0, 5.3.0, 5.8.0
Camel	Enterprise Integration Framework	420-4937	516-1935	1271	6.4%-37.79%	1.4.0, 2.9.0, 2.10.0, 2.11.0
Derby	Relational Database	640-1155	207-1933	4362	33.37%-52.47%	10.2.1.6, 10.3.1.4, 10.5.1.1
Groovy	Java-syntax-compatible OOP for JAVA	137-345	93-245	382	18.28%-29.73%	1.5.7, 1.6.0.Beta1, 1.6.0.Beta 2
HBase	Distributed Scalable Data Store	536-942	540-900	2632	35.74%-38.67%	0.94.0, 0.95.0, 0.95.2
Hive	Data Warehouse System for Hadoop	245-524	266-1113	767	15.27%-41.73%	0.9.0, 0.10.0, 0.12.0
JRuby	Ruby Programming Lang for JVM	687-2674	318-501	637	13.77%-36.16%	1.1, 1.4, 1.5, 1.7
Lucene	Text Search Engine Library	195-792	299-1080	1387	6.57%-59.2%	2.3.0, 2.9.0, 3.0.0, 3.1.0
Wicket	Web Application Framework	1-1370	735-782	479	8.3%-13.14%	1.3.0.beta1, 1.3.0.beta2, 1.5.3

command, following the procedure adopted in previous studies [6], [33].

Recent studies have cautioned that range-based strategies are more accurate than time-based ones for assigning commits to releases [74], [75]. Hence, we employed a range-based approach to minimize errors in commit-to-release mapping. After obtaining the release-wise commit data, we computed all conventional process metrics as defined by Rahman and Devanbu [4] for the source code files of all releases and for all projects included in our dataset. This set of 14 widely used process metrics, as defined by Rahman and Devanbu [4], has been adopted in replication and benchmarking studies on defect prediction [6] and is commonly used for evaluating process-based defect predictors. Following recent methodological recommendations [76], [77], we excluded untouched modules—files that were never modified during the studied releases—as their inclusion can bias the results. For such modules, product metrics remain static while process metrics lack meaningful values. Consequently, our dataset includes only those source code files that were modified at least once during the analyzed releases.

For each source code file modified in release R_j, we obtain its product metric values and post-release defect label from the dataset corpus of Yatish et al. In contrast, the conventional process metrics defined by Rahman and Devanbu [4], along with their size-aware vector counterparts, are computed by us. Because commits that modify more than 100 files are excluded, each process-metric vector has 100 components, where the i^{th} component aggregates the contribution from commits of size i. As an example, for the scalar *Commit Count* metric, we construct a *Commit Count Vector* for each file f, where the i^{th} component represents the proportion of commits that modified f and had a size of i. For brevity, we do not enumerate the product metrics or the scalar process metrics here; the scalar process metrics are detailed in Rahman and Devanbu [4], and the product metrics used in this study are described by Yatish et al. [49]. Table II illustrates how each vector process metric is derived from its corresponding scalar counterpart. Since each of the 14 scalar process metrics is expanded into a 100-dimensional vector form, this yields a total of 1,400 vector process metrics. In summary, our feature set comprises 54 product metrics, 14 scalar process metrics, and 1,400 vector process metrics used for classification.

2) Evaluation Protocol: For each project release, we collect all files changed in that release and obtain their product metrics, scalar process metrics, and size-aware vector process metrics as described above. We also extract each file's post-release defect label. However, not all metrics were used to train the classifiers. Before model training, we performed feature selection using the HSIC Lasso method. The HSIC Lasso requires specifying the number of top features (k) to be selected. We experimented with different values of k and observed that the best performance was obtained when $k = 40$. Consequently, we selected the top 40 features from each feature set combination for all subsequent experiments.

We evaluate performance using bootstrap validation with 100 out-of-sample resamples, per project and per classifier. Across 9 projects and 5 classifiers, this yields 45 project–classifier configurations. This is in accordance with past studies [49]. For each configuration, we train and test two feature sets: (i) Product + Scalar Process (PR+SP) and (ii) Product + Vector Process (PR+VP). If PR+VP outperforms PR+SP, we infer that vector process metrics add value for defect classification.

To assess performance comprehensively, we report F1, AUROC, MCC, Brier score, and area under the precision–recall curve (AUPRC). We do not report accuracy, since defect datasets are typically imbalanced and accuracy is not appropriate in such settings [54], [78]. F1 balances precision and recall for the minority (buggy) class; MCC summarizes all four confusion matrix terms and is robust under imbalance [79]; AUROC measures ranking ability across thresholds; AUPRC emphasizes performance on the positive class; and the Brier score evaluates the calibration of predicted probabilities. These measures are widely used in defect prediction studies [28], [49].

E. Experimental Setup to Answer RQ2

1) Details of the Metrics: Along with the product metrics, scalar process metrics, and vector process metrics discussed in Section III-D, a fourth category of metrics used in this study is derived from the *co-change graph*. Traditionally, a simple co-change graph connects two source code files if they are modified together in the same commit. However, such pairwise graphs fail to encode commit-size information—i.e., whether the files co-occurred in small or large commits.

To overcome this limitation, we construct a *hyper co-change graph* from the commit history, where nodes represent source code entities and each hyperedge links all files modified together in a commit. Unlike pairwise networks, where edges

TABLE II: Vector Process Metrics

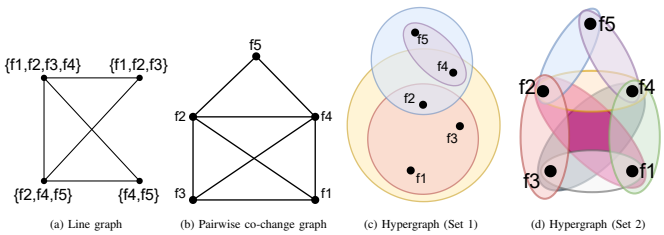
Metric	Definition	Formula used to compute i^{th} component for the vector	Notations
Commit Count Vector ($\mathbf{V}_f^{(comm)}$)	Distribution of how often file f appears in commits of different sizes.	$\mathbf{V}_f^{(\text{comm})}[i] = \frac{\lvert C_{f,i}\rvert}{\lvert C_f\rvert}$	$C_{f,i}$ = commits of size i involving file f; C_f = total commits involving f.
Developer Activity Vector ($\mathbf{V}_f^{(adev)}$)	Distribution of the number of developers who modified file f across commits of different sizes in the target release.	$\mathbf{V}_f^{(\text{adev})}[i] = \frac{\lvert D_{f,i}\rvert}{\lvert D_f\rvert}$	$D_{f,i}$ = developers of commits of size i involving file f in target release; D_f = developers of all commits involving f in target release
Distinct Developer Activity Vector ($\mathbf{V}_f^{(ddev)}$)	Distribution of the number of distinct developers who modified file f during the project's lifetime across commits of different sizes up to the target release.	$\mathbf{V}_f^{(\text{ddev})}[i] = \frac{\lvert D^*_{f,i}\rvert}{\lvert D^*_f\rvert}$	$D^*_{f,i}$ = developers of commits of size i involving file f during the project's lifetime up to the target release; D^*_f = developers of all commits involving f in the project up to the target release.
Lines Added Vector ($\mathbf{V}_f^{(add)}$)	Distribution of lines added to f across commit sizes.	$\mathbf{V}_f^{(\text{add})}[i] = \frac{\lvert LA_{f,i}\rvert}{\lvert LA_f\rvert}$	$\text{LA}_{f,i}$ = lines added to f in commits of size i; LA_f = lines added to file f across all commits involving it
Lines Deleted Vector ($\mathbf{V}_f^{(del)}$)	Distribution of lines deleted from f across commit sizes.	$\mathbf{V}_f^{(\text{del})}[i] = \frac{\lvert LD_{f,i}\rvert}{\lvert LD_f\rvert}$	$\text{LD}_{f,i}$ = lines deleted to f in commits of size i; LD_f = lines deleted to file f across all commits involving it
Owner Contribution Vector ($\mathbf{V}_f^{(own)}$)	Owner's contribution distribution across commit sizes.	$\mathbf{V}_f^{(\text{own})}[i] = \frac{\lvert L^{(o)}_{f,i}\rvert}{\lvert L^{(o)}_f\rvert}$	$\text{L}^{(\text{o})}_{f,i}$ = Lines changed by the owner in file f through commits of size i in the target release; $\text{L}^{(\text{o})}_f$ = lines changed by the owner in all commits involving f in the target release.
Owner Experience Vector ($\mathbf{V}_f^{(oexp)}$)	Distribution of owner's experience in the project across commit sizes .	$\mathbf{V}_f^{(\text{oexp})}[i] = \frac{\lvert L^{(o*)}_{f,i}\rvert}{\lvert L^{(o*)}_f\rvert}$	$\text{L}^{(\text{o}*)}_{f,i}$ = lines changed by the owner of f in commits of size i in project lifetime till target release; $\text{L}^{(\text{o}*)}_f$ = lines changed by the owner in all commits in project lifetime till target release
Developers' Experience Vector ($\mathbf{V}_f^{(exp)}$)	Represents the average experience of authors contributing to f across the commit sizes.	$\mathbf{V}_f^{(\text{exp})}[i] = G_{\text{Mean}}\{E_D\}$	E_D = set of $\frac{e(d_{j,i})}{e(d_j)}$ where, $e(d_{j,i})$ = experience of developer j who changed file f through commits of size i; $e(d_j)$ = experience of the developer j who changed file f.
Minor Contributors Vector ($\mathbf{V}_f^{(minor)}$)	Distribution of minor developers across the commit sizes .	$\mathbf{V}_f^{(\text{minor})}[i] = \frac{\lvert M_{f,i}\rvert}{\lvert M_f\rvert}$	$M_{f,i}$ = minor developers considering commits of size i involving f; M_f = minor developers considering all commits involving f.
Size-aware change Entropy Vector (SCTR) ($\mathbf{V}_f^{(sctr)}$)	commit size aware Change Entropy Vector f	$\mathbf{V}_f^{(\text{sctr})}[i] = p_i(f).H_i$	$p_i(f)$ is the proportion of commits of size i involving f; H_i is the entropy of considering commit of size i across all files
Neighborhood Commit count Vector (NCOMM) ($\mathbf{V}_f^{(ncomm)}$)	Aggregates neighbors' commit count vector information.	$\frac{\sum_n w_n \mathbf{V}_n^{(comm)}}{\sum_n w_n \lVert \mathbf{V}_n^{(comm)}\rVert_1}$	$n \in$ neighbors of f in the co-change graph; w_n = edge weight between f and n in co-change graph.
Neighborhood Developer Activity Vector (NADEV) ($\mathbf{V}_f^{(nadev)}$)	Aggregates neighbors' Developer Activity Vector information.	$\frac{\sum_n w_n \mathbf{V}_n^{(adev)}}{\sum_n w_n \lVert \mathbf{V}_n^{(adev)}\rVert_1}$	$n \in$ neighbors of f in the co-change graph; w_n = edge weight between f and n in co-change graph.
Neighborhood Distinct Developer Vector (NDDEV) ($\mathbf{V}_f^{(nddev)}$)	Like NADEV, but using distinct author vectors of neighbors.	$\frac{\sum_n w_n \mathbf{V}_n^{(ddev)}}{\sum_n w_n \lVert \mathbf{V}_n^{(ddev)}\rVert_1}$	$n \in$ neighbors of f in the co-change graph; w_n = edge weight between f and n in co-change graph.
Neighborhood Entropy Vector (NSCTR) ($\mathbf{V}_f^{(nsctr)}$)	Aggregating Neighbors change-entropy vector information.	$\frac{\sum_n w_n \mathbf{V}_n^{(sctr)}}{\sum_n w_n \lVert \mathbf{V}_n^{(sctr)}\rVert_1}$	$n \in$ neighbors of f in the co-change graph; w_n = edge weight between f and n in co-change graph.

connect only two nodes, hypergraphs allow hyperedges connecting multiple nodes simultaneously, thereby preserving the group-wise semantics of co-changes.

Table III presents a sample change history, showing the files and the commits in which they were modified. In a simple pairwise co-change graph, two file nodes are connected if they were changed in the same commit, whereas in a hyper co-change graph, all files changed together in a commit form a single hyperedge. As illustrated in Figure 3 and Table III, two commit sets of markedly different nature can produce identical pairwise co-change graphs (Figure 3b) but distinct hypergraphs (Figures 3c and 3d) that retain commit-size information. Consequently, centrality measures on pairwise co-change graphs cannot distinguish a file's importance arising from participation in large versus small commits—an important distinction naturally captured by the hypergraph formulation.

Files	F1	F2	F3	F4	F5
Commit Set 1	*C1, C4*	*C1, C2, C4*	*C1, C4*	*C1, C2, C3*	*C2, C3*
Commit Set 2	*C1, C5, C6*	*C1, C7, C8*	*C3, C4*	*C2, C3, C5, C7*	*C2, C4, C6, C8*

TABLE III: Example commit sets used for the network comparison in Figure 3.



(a) Line graph (b) Pairwise co-change graph (c) Hypergraph (Set 1) (d) Hypergraph (Set 2)

Fig. 3: Hyper co-change vs. pairwise co-change networks in capturing commit size. Pairwise graphs collapse group edits into binary links, obscuring commit size, while hypergraphs preserve the full change set via hyperedges.

For node importance measures in hypergraph we adopt the approach proposed by Kovalenko et al. [30], which introduces *vector centralities* for hypergraphs, generalizing classical centrality measures to higher-order interactions.

Vector centrality extends standard SNA metrics—such as degree, betweenness, closeness, and eigenvector centrality—to hypergraphs, where interactions can involve more than two nodes simultaneously. Let $G = (N, \mathcal{M})$ be an undirected hypergraph, where N is the set of nodes and $\mathcal{M} \subseteq 2^N$ is the set of hyperedges (non-empty subsets of N). Let $H = \max_{e \in \mathcal{M}} |e|$ be the maximum cardinality (or order) of any hyperedge.

To compute vector centralities, we construct the *line graph* (also called the *hyperedge adjacency graph*) of the hypergraph G, denoted $I(G)$. In this graph, each node represents a hyperedge $e \in \mathcal{M}$, and an undirected edge is added between two hyperedges $e, e' \in \mathcal{M}$ if they share at least one common node (i.e., $e \cap e' \neq \emptyset$). This line graph encodes the pairwise overlap structure among hyperedges, enabling the application of classical centrality measures to hyperedges themselves. In our example, for Change Set 1 shown in Table III and the corresponding hypergraph depicted in Figure 3c, the associated line graph is shown in Figure 3a.

Let $x(e)$ denote the centrality score of a hyperedge $e \in \mathcal{M}$ as computed on $I(G)$ using a classical centrality measure (e.g., eigenvector, closeness, betweenness, or degree centrality). The l-th component of the vector centrality for node $j \in N$ is defined as:

$$x_{j,l} = \frac{1}{l} \sum_{\substack{e \in \mathcal{M} \\ |e|=l \\ j \in e}} x(e), \tag{1}$$

for $l = 2, 3, \ldots, H$. This expression quantifies the importance of a node within the hypergraph by considering its participation in hyperedges of size l. Specifically, it captures how central a node is when interacting with other nodes through group associations of order l. In the context of vector centrality, this component-level view allows us to disentangle a node's role across different interaction orders, rather than collapsing all contributions into a single scalar score. By normalizing the contribution of each hyperedge by its size, the measure avoids overemphasizing large hyperedges and provides a more balanced perspective on the node's involvement in group-level interactions of a specific order. It has been shown that $\sum_{j \in N} \|\vec{x}_j\|_1 = 1$ making it a properly normalized measure.

It should be noted that we compute vector centralities for each node corresponding to the global centrality measures commonly used in pairwise networks—namely, degree, betweenness, closeness, and eigenvector centralities. Since we filter out commits that modify more than 100 files, the maximum number of components in the vector centrality of a node for a given centrality type (e.g., betweenness) is 100. Considering all four centrality measures, the hypergraph-based vector centralities yield a total of 400 features for each node (source code file).

2) Evaluation Protocol: To answer RQ2, we conduct experiments across all project–classifier pairs using the same machine learning pipeline employed for RQ1. The objective here is to examine whether incorporating vector centrality further enhances defect prediction performance. For this purpose, we evaluate classifiers using the metric set *Product + Vector Process Metrics + Vector Centrality Scores* (PR+VP+VC), applying the same evaluation metrics as in RQ1.

Our primary focus is to determine whether the performance gain achieved by PR+VP+VC over PR+SP (scalar process metrics) exceeds the improvement observed when using PR+VP over PR+SP. To assess this, we first compute how often (out of 45 project–classifier combinations) each metric set—PR+SP, PR+VP, and PR+VP+VC—achieves the best performance for each evaluation measure (e.g., AUROC, MCC). However, simple frequency counts do not establish statistical confidence. Therefore, following the past studies [28], we apply the Friedman test to determine whether the performance differences among the metric sets are statistically significant, followed by the post-hoc Nemenyi test to identify pairwise rankings. This analysis is conducted using the dataset-classifier pair results for all the metrics considered in this study. Performing statistical validation across dataset-classifier pairs helps eliminate biases in the conclusions drawn from machine learning experiments, as also emphasized in recent studies within the machine learning domain [80], [81].

In practice, the ability of defect classifiers to prioritize the most defect-prone files is often more relevant. Hence, we also evaluate the top-k ranking performance by ordering source code files according to their predicted probability of being defective and analyzing the resulting Precision@k and Recall@k curves.

IV. Results and Discussion

A. Answer to RQ1

To answer RQ1, we trained our classifiers using two feature sets: Product + Scalar Process (PR+SP) and Product + Vector Process (PR+VP) metrics. The experiments were conducted across 45 dataset–classifier pairs (9 projects $\times$ 5 classifiers), and the aggregated results across projects are shown as box plots in Fig. 4. The figure shows that PR+VP

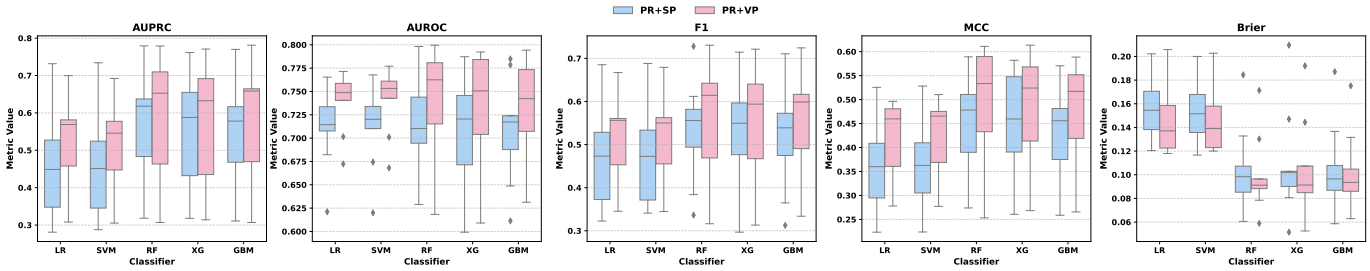


Fig. 4: Absolute performance of PR+SP and PR+VP feature combinations across multiple metrics

consistently outperforms PR+SP across all classifiers and evaluation metrics. The distributions of PR+VP scores are shifted in the favorable direction, exhibiting higher medians for AUROC, AUPRC, F1, and MCC, and lower medians for Brier scores, indicating systematic performance gains rather than improvements restricted to a few outliers. These results confirm that incorporating commit-size semantics into process metrics enhances prediction accuracy, ranking ability, and calibration of defect prediction models.

The median AUPRC improves by 5.6%–26.7% (maximum with Logistic Regression), suggesting a higher proportion of true defects retrieved among top-ranked files under class imbalance. AUROC increases by 3.5%–7.3% (maximum with Random Forest), reflecting improved discrimination between defective and clean files across thresholds. F1 rises by 8.0%–17.6% (maximum with Logistic Regression), demonstrating a better precision–recall balance. MCC improves by 11.5%–28.3% (maximum with SVM), indicating stronger agreement between predictions and actual outcomes. The Brier score decreases by 3.0%–11.3% (maximum with Logistic Regression), confirming better probability calibration and reliability for risk-based decisions.

Although no single classifier dominates across all metrics, the consistent distributional shifts and significant median gains clearly demonstrate that integrating commit-size semantics through vector process metrics (PR+VP) produces more discriminative, balanced, and well-calibrated defect prediction models than scalar process metrics (PR+SP). In a nutshell, the answer to RQ1 is as follows:

Answer to RQ1: *Incorporating commit-size semantics through vector process metrics (PR+VP) significantly enhances the effectiveness of product metrics compared to scalar process metrics (PR+SP). Median improvements range between 3.5%–7.3% for AUROC, 5.6%–26.7% for AUPRC, 8.0%–17.6% for F1, and 11.5%–28.3% for MCC, while Brier scores decrease by 3.0%–11.3%. These consistent gains confirm that commit-size–aware process representations yield more discriminative, better-calibrated, and balanced defect prediction models.*

B. Answer to RQ2

To answer RQ2, we trained classifiers using the combined feature set PR+VP+VC, which integrates product metrics, vector process metrics, and vector centrality measures. To assess whether incorporating vector centrality features yields additional benefits beyond PR+VP, we computed the evaluation metrics (e.g., AUROC) for each dataset–classifier pair and analyzed the percentage improvements relative to the baseline PR+SP configuration. Specifically, we compared the improvements achieved by PR+VP over PR+SP and those achieved by PR+VP+VC over PR+SP.

The results are presented in Figure 5. Each box plot depicts the percentage improvement over the PR+scalar process baseline (PR+SP) across projects for each classifier, comparing two configurations: PR+VP and PR+VP+VC. The gray line within each box represents the median (Q2), while the red line indicates the mean. For metrics such as AUROC, F1, AUPRC and MCC—where higher values imply better performance—the median and mean percentage gains of PR+VP+VC consistently exceed those of PR+VP. In several classifiers, the entire interquartile range (IQR) of the PR+VP+VC boxes shifts upward, indicating both higher and more stable performance improvements across projects. This clearly demonstrates that incorporating vector centrality features further enhances model performance on both rank- and threshold-based measures.

For the Brier score, where lower values indicate better calibration, the boxes are centered around negative percentage differences relative to PR+SP. Here, PR+VP+VC exhibits more negative median and mean values than PR+VP, reflecting a greater reduction in Brier score and consequently better model calibration.

As evident from Figures 4 and 5, the PR+VP+VC configuration consistently outperforms both PR+VP and PR+SP, confirming the overall utility of size-aware process metrics. Since the numerical differences among these feature sets appear moderate, it is essential to assess whether these improvements are statistically significant. To this end, we applied the Friedman test across all dataset–classifier pairs for the three feature sets (PR+SP, PR+VP, and PR+VP+VC). The results yielded a p-value < 0.05, indicating significant differences among their performance scores.

To further identify which feature set performs best, we conducted a post-hoc Nemenyi test. The results, presented in Figure 6 reveal that except for the Brier score, the improvements of PR+VP over PR+SP are statistically significant across all other four metrics. Moreover, the mean rank differences between PR+VP+VC and both PR+VP and PR+SP

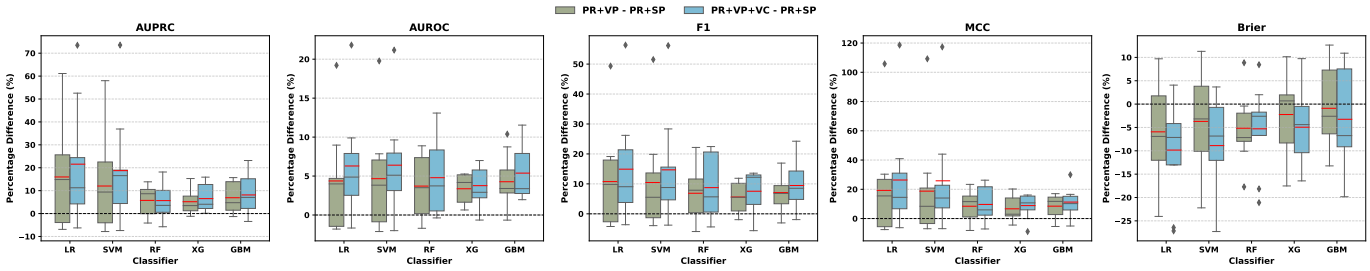
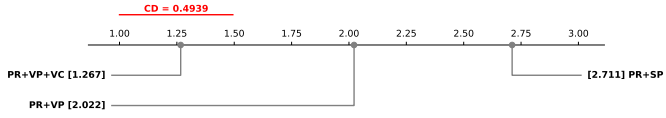
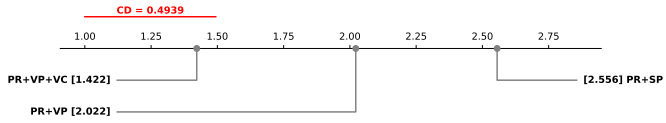


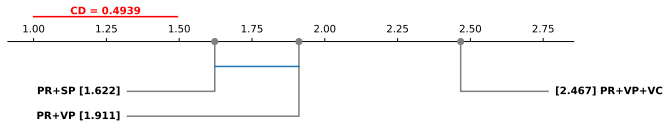
Fig. 5: Comparison of percentage improvements across performance metrics from PR+SP to PR+VP, and from PR+SP to PR+VP+VC



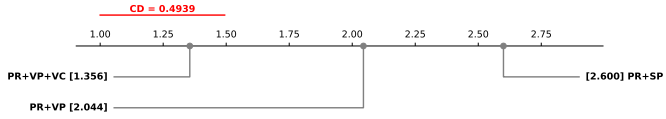
(a) AUROC CD plot



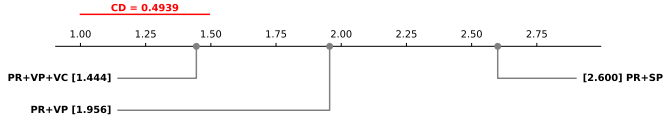
(b) MCC CD plot



(c) Brier CD plot



(d) F1 CD plot



(e) AUPRC CD plot

Fig. 6: Critical Difference plots showing the results of the post-hoc Nemenyi test

exceed the critical distance in the CD plots, confirming that the inclusion of vector centrality features provides statistically significant gains across all five evaluation metrics. Overall, the Friedman–Nemenyi tests demonstrate that PR+VP+VC achieves the highest ranks for all classification performance measures, outperforming both baselines.

The pie chart in Figure 9 summarizes the frequency with which each feature set achieved the best performance across 45 dataset–classifier combinations. PR+VP+VC achieved the highest proportion of best performances—75.6% in AUROC, 66.7% in MCC, 71.1% in F1, 62.2% in AUPRC, and lowest Brier scores in 51.1% of cases. The clear and significant wider gap between PR+VP+VC and PR+SP in the CD plots, along with its consistently lower Brier scores, underscores the added confidence and calibration achieved by incorporating vector centrality features.

Since one of the key objectives in defect prediction is to rank source code files according to their defect proneness, we also evaluated the models from a ranking perspective. We assessed the top-k performance of all three feature sets using mean *precision@k* (mean across the projects) and *recall@k* for $k =$ 1 to 100, as shown in Figures 7 and 8. The results indicate that although PR+VP+VC performs best overall, its *precision@k* and *recall@k* values are only marginally higher than those of PR+VP. However, both size-aware configurations (PR+VP and PR+VP+VC) substantially outperform PR+SP across all k values.

In summary, the results confirm that incorporating vector centrality features over the hyper co-change graph significantly improves defect classification performance and ranking quality. Among all three feature sets—PR+SP, PR+VP, and PR+VP+VC—the PR+VP+VC configuration consistently delivers the best and most reliable outcomes.

Answer to RQ2: *Incorporating vector centrality features over the hyper co-change graph (PR+VP+VC) yields statistically significant improvements over both PR+VP and PR+SP. The Friedman test and post-hoc Nemenyi test confirm that PR+VP+VC consistently achieves the best mean ranks across all five evaluation metrics. Across all 45 dataset–classifier pairs, PR+VP+VC achieves the highest frequency of best performance—75.6% in AUROC, 66.7% in MCC, 71.1% in F1, 62.2% in AUPRC, and lower Brier scores in 51.1% of cases. Both PR+VP and PR+VP+VC also outperform PR+SP on ranking-based metrics (precision@k, recall@k) for all* $k \in [1, 100]$. *Overall, the addition of vector centrality features enhances both discriminative and calibration capabilities of defect classifiers, establishing PR+VP+VC as the most effective and statistically superior feature set.*

V. Threats to Validity

There are several potential threats to the validity of this study. First, although the experiments were conducted on a

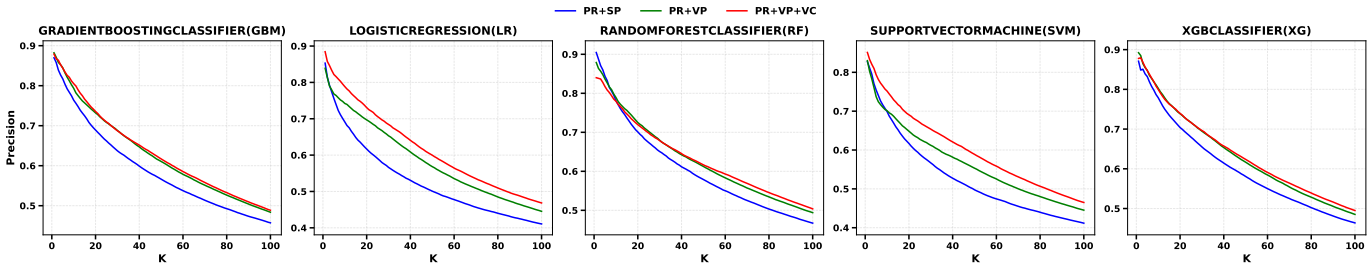


Fig. 7: Mean Precision at k for different feature combinations across classifiers

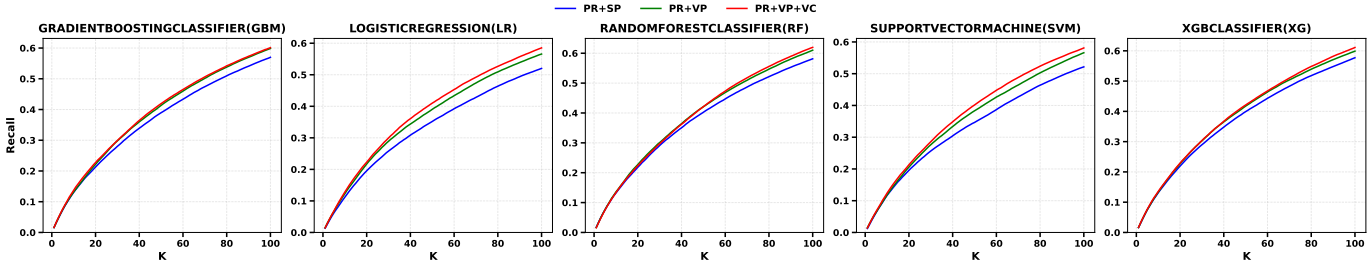


Fig. 8: Mean Recall at k for different feature combinations across classifiers

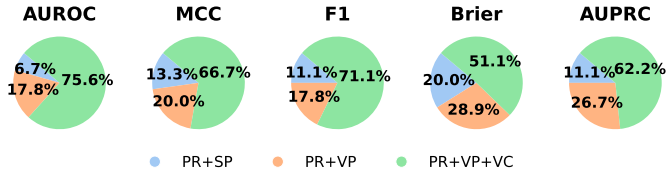


Fig. 9: Pie chart showing the percentage of times each feature combination yields the best performance

widely used dataset in software engineering research, it includes only Java projects, which may limit the generalizability of the results. Second, we relied on a single dataset to evaluate our approach; experiments on additional and more diverse datasets would further strengthen the generalizability of the findings. Finally, the combined feature set comprising product (PR), vector process (VP), and vector centrality (VC) metrics results in a high-dimensional space of approximately 1,800 features. Although we employ a scalable feature selection technique (HSIC Lasso) and train classifiers on the top 40 selected features, the overall model remains computationally expensive.

VI. Conclusion

This study proposed two novel approaches to enhance file-level software defect prediction by integrating commit-size semantics and higher-order co-change relationships into process metrics and network-based representations. First, we redefined conventional scalar process metrics into *commit-size–aware vector process metrics* that capture how a file's change behavior is distributed across commits of different magnitudes. Extensive experiments on nine open-source projects demonstrated that these vectorized metrics, when combined with product metrics, substantially improve prediction accuracy, calibration, and ranking performance over traditional scalar process metrics. Second, we modeled co-change relationships among files as a *hyper co-change network* and employed *vector centrality* measures to quantify file importance across commits of varying sizes. Integrating these hypergraph-based centralities with product and vector process metrics yielded statistically significant performance gains across multiple classifiers and evaluation metrics, as confirmed by Friedman and Nemenyi tests. Overall, the findings establish that commit-size semantics—captured through vector process metrics and hypergraph vector centralities—provide a richer and more discriminative characterization of software change behavior. These insights open avenues for further research on size-aware and higher-order representations in predictive software analytics.

Data Availability

To promote transparency and reproducibility, we provide a replication package[2] containing the code, dataset, and detailed instructions to replicate the experiments.

References

[1] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[2] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[3] A. E. Hassan, "Predicting faults using the complexity of code changes," in *2009 IEEE 31st international conference on software engineering*, IEEE. USA: IEEE Computer Society, 2009, pp. 78–88.

[4] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *2013 35th international conference on software engineering (ICSE)*, IEEE. San Francisco, CA, USA: IEEE Press, 2013, pp. 432–441.

[2]https://github.com/unpaidresearcher/SANER2026

[5] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy, "Does distributed development affect software quality? an empirical case study of windows vista," *Communications of the ACM*, vol. 52, no. 8, pp. 85–93, 2009.

[6] S. Majumder, P. Mody, and T. Menzies, "Revisiting process versus product metrics: a large scale analysis," *Empirical Software Engineering*, vol. 27, no. 3, p. 60, 2022.

[7] M. Hammad, M. L. Collard, and J. I. Maletic, "Automatically identifying changes that impact code-to-design traceability during evolution," *Software Quality Journal*, vol. 19, no. 1, pp. 35–64, 2011.

[8] C. Le Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *Software quality journal*, vol. 21, no. 3, pp. 421–443, 2013.

[9] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, "Studying re-opened bugs in open source software," *Empirical Software Engineering*, vol. 18, no. 5, pp. 1005–1042, 2013.

[10] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *IEEE transactions on software engineering*, vol. 27, no. 1, pp. 1–12, 2002.

[11] A. Bandi, B. J. Williams, and E. B. Allen, "Empirical evidence of code decay: A systematic mapping study," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 341–350.

[12] Z. Li and J. Long, "A case study of measuring degeneration of software architectures from a defect perspective," in *2011 18th Asia-Pacific Software Engineering Conference*. IEEE, 2011, pp. 242–249.

[13] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

[14] C. Ni, X. Xia, D. Lo, X. Yang, and A. E. Hassan, "Just-in-time defect prediction on javascript projects: A replication study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–38, 2022.

[15] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[16] C. Khanan, W. Luewichana, K. Pruktharathikoon, J. Jiarpakdee, C. Tantithamthavorn, M. Choetkiertikul, C. Ragkhitwetsagul, and T. Sunetnanta, "Jitbot: an explainable just-in-time defect prediction bot," in *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, 2020, pp. 1336–1339.

[17] Y. Fan, X. Xia, D. A. Da Costa, D. Lo, A. E. Hassan, and S. Li, "The impact of mislabeled changes by szz on just-in-time defect prediction," *IEEE transactions on software engineering*, vol. 47, no. 8, pp. 1559–1586, 2019.

[18] A. T. Misirli, E. Shihab, and Y. Kamei, "Studying high impact fix-inducing changes," *Empirical Software Engineering*, vol. 21, no. 2, pp. 605–641, 2016.

[19] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[20] M. D'Ambros, M. Lanza, and R. Robbes, "On the relationship between change coupling and software defects," in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 135–144.

[21] L. L. Silva, M. T. Valente, and M. A. Maia, "Co-change patterns: A large scale empirical study," *Journal of Systems and Software*, vol. 152, pp. 196–214, 2019.

[22] E. Hrishikesh, A. Kumar, M. Bhardwaj, and S. Agarwal, "Co-change graph entropy: A new process metric for defect prediction," *arXiv preprint arXiv:2504.18511*, 2025.

[23] E. Kouroshfar, "Studying the effect of co-change dispersion on software quality," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1450–1452.

[24] S. Kirbas, B. Caglayan, T. Hall, S. Counsell, D. Bowes, A. Sen, and A. Bener, "The relationship between evolutionary coupling and defects in large industrial software," *Journal of Software: Evolution and Process*, vol. 29, no. 4, p. e1842, 2017.

[25] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 531–540.

[26] T. H. Nguyen, B. Adams, and A. E. Hassan, "Studying the impact of dependency network measures on software quality," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[27] S. Amasaki, "Cross-version defect prediction: use historical data, cross-project data, or both?" *Empirical Software Engineering*, vol. 25, pp. 1573–1595, 2020.

[28] L. Gong, G. K. Rajbahadur, A. E. Hassan, and S. Jiang, "Revisiting the impact of dependency network metrics on software defect prediction," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 5030–5049, 2021.

[29] G. Lee, F. Bu, T. Eliassi-Rad, and K. Shin, "A survey on hypergraph mining: Patterns, tools, and generators," *ACM Computing Surveys*, vol. 57, no. 8, pp. 1–36, 2025.

[30] K. Kovalenko, M. Romance, E. Vasilyeva, D. Aleja, R. Criado, D. Musatov, A. M. Raigorodskii, J. Flores, I. Samoylenko, K. Alfaro-Bittner *et al.*, "Vector centrality in hypergraphs," *Chaos, Solitons & Fractals*, vol. 162, p. 112397, 2022.

[31] S. Majumder, J. Chakraborty, and T. Menzies, "When less is more: on the value of "co-training" for semi-supervised software defect predictors," *Empirical Software Engineering*, vol. 29, no. 2, p. 51, 2024.

[32] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change bursts as defect predictors," in *2010 IEEE 21st international symposium on software reliability engineering*, IEEE. USA: IEEE Computer Society, 2010, pp. 309–318.

[33] M. Wen, R. Wu, and S.-C. Cheung, "How well do change sequences predict defects? sequence learning from software changes," *IEEE Transactions on Software Engineering*, vol. 46, no. 11, pp. 1155–1175, 2018.

[34] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 181–190.

[35] E. Kouroshfar, M. Mirakhorli, H. Bagheri, L. Xiao, S. Malek, and Y. Cai, "A study on the role of software architecture in the evolution and quality of software," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 246–257.

[36] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan, "High-impact defects: a study of breakage and surprise defects," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 300–310.

[37] M. Yan, Y. Fang, D. Lo, X. Xia, and X. Zhang, "File-level defect prediction: Unsupervised vs. supervised models," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2017, pp. 344–353.

[38] L. Šikić, P. Afrić, A. S. Kurdija, and M. ŠIlić, "Improving software defect prediction by aggregated change metrics," *IEEE access*, vol. 9, pp. 19 391–19 411, 2021.

[39] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 13–23.

[40] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008, pp. 2–12.

[41] L. L. Silva, M. T. Valente, and M. de A. Maia, "Co-change clusters: Extraction and application on assessing software modularity," *Transactions on Aspect-Oriented Software Development XII*, pp. 96–131, 2015.

[42] L. L. Silva, M. T. Valente, and M. d. A. Maia, "Assessing modularity using co-change clusters," in *Proceedings of the 13th international conference on Modularity*, 2014, pp. 49–60.

[43] D. Cui, J. Wang, Q. Wang, P. Ji, M. Qiao, Y. Zhao, J. Hu, L. Wang, and Q. Li, "Three heads are better than one: suggesting move method refactoring opportunities with inter-class code entity dependency enhanced hybrid hypergraph neural network," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 745–757.

[44] L. Wang, Q. Wang, J. Wang, Y. Zhao, M. Wei, Z. Quan, D. Cui, and Q. Li, "Hecs: A hypergraph learning-based system for detecting extract class refactoring opportunities," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1851–1855.

[45] G. Rong, Y. Zhang, L. Yang, F. Zhang, H. Kuang, and H. Zhang, "Modeling review history for reviewer recommendation: A hypergraph approach," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 1381–1392.

[46] E. Ersoy, K. Kaya, M. Altınışık, and H. Sözer, "Using hypergraph clustering for software architecture reconstruction of data-tier software," in *European Conference on Software Architecture*. Springer, 2016, pp. 326–333.

[47] J. Jiang, L. Ren, Y. Xiong, and L. Zhang, "Inferring program transformations from singular examples via big code," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 255–266.

[48] S. Qiu, M. Huang, Y. Liang, C. Peng, and Y. Yuan, "Code multiview hypergraph representation learning for software defect prediction," *IEEE Transactions on Reliability*, vol. 73, no. 4, pp. 1863–1876, 2024.

[49] S. Yatish, J. Jiarpakdee, P. Thongtanunam, and C. Tantithamthavorn, "Mining software defects: Should we consider affected releases?" in *2019 IEEE/ACM 41st international conference on software engineering (ICSE)*, IEEE. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 654–665.

[50] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto, "Predicting defective lines using a model-agnostic technique," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1480–1496, 2020.

[51] G. Lee and S. U.-J. Lee, "An empirical comparison of model-agnostic techniques for defect prediction models," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 179–189.

[52] J. Jiarpakdee, C. K. Tantithamthavorn, H. K. Dam, and J. Grundy, "An empirical study of model-agnostic techniques for defect prediction models," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 166–185, 2020.

[53] Z. Li, M. Pan, Y. Pei, T. Zhang, L. Wang, and X. Li, "Robust learning of deep predictive models from noisy and imbalanced software engineering datasets," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[54] R. Moussa and F. Sarro, "On the use of evaluation measures for defect prediction studies," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 101–113.

[55] P. Thongtanunam and C. Tantithamthavorn, "Code ownership: The principles, differences, and their associations with software quality," in *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2024, pp. 379–390.

[56] P. Mahbub, O. Shuvo, and M. M. Rahman, "Defectors: A large, diverse python dataset for defect prediction," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 2023, pp. 393–397.

[57] X. Zhou, D. Han, and D. Lo, "Bridging expert knowledge with deep learning techniques for just-in-time defect prediction," *Empirical Software Engineering*, vol. 30, no. 1, pp. 1–44, 2025.

[58] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 560–560.

[59] H. Keshavarz and M. Nagappan, "Apachejit: a large dataset for just-in-time defect prediction," in *Proceedings of the 19th international conference on mining software repositories*, 2022, pp. 191–195.

[60] 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*, 2022, pp. 672–683.

[61] S. Pan, L. Bao, X. Xia, D. Lo, and S. Li, "Fine-grained commit-level vulnerability type prediction by cwe tree structure," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 957–969.

[62] L. Gong, H. Zhang, J. Zhang, M. Wei, and Z. Huang, "A comprehensive investigation of the impact of class overlap on software defect prediction," *IEEE transactions on software engineering*, vol. 49, no. 4, pp. 2440–2458, 2022.

[63] J. Shin, R. Aleithan, J. Nam, J. Wang, and S. Wang, "Explainable software defect prediction: Are we there yet?" *arXiv preprint arXiv:2111.10901*, 2021.

[64] G. K. Rajbahadur, S. Wang, Y. Kamei, and A. E. Hassan, "The impact of using regression models to build defect classifiers," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 135–145.

[65] S. Wang, L. Huang, A. Gao, J. Ge, T. Zhang, H. Feng, I. Satyarth, M. Li, H. Zhang, and V. Ng, "Machine/deep learning for software engineering: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 49, no. 3, pp. 1188–1231, 2022.

[66] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[67] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 1–18, 2016.

[68] J. Jiarpakdee, C. Tantithamthavorn, and A. E. Hassan, "The impact of correlated metrics on the interpretation of defect models," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 320–331, 2019.

[69] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[70] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 434–443, 2013.

[71] Z. Xu, J. Liu, Z. Yang, G. An, and X. Jia, "The impact of feature selection on defect prediction performance: An empirical comparison," in *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*. IEEE, 2016, pp. 309–320.

[72] J. Jiarpakdee, C. Tantithamthavorn, and C. Treude, "Autospearman: Automatically mitigating correlated software metrics for interpreting defect models," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 92–103.

[73] M. Yamada, W. Jitkrittum, L. Sigal, E. P. Xing, and M. Sugiyama, "High-dimensional feature selection by feature-wise kernelized lasso," *Neural computation*, vol. 26, no. 1, pp. 185–207, 2014.

[74] F. C. do Rego Pinto, B. Costa, and L. Murta, "Assessing time-based and range-based strategies for commit assignment to releases," in *2021 IEEE international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2021, pp. 142–153.

[75] F. C. d. R. Pinto and L. G. P. Murta, "On the assignment of commits to releases," *Empirical Software Engineering*, vol. 28, no. 2, p. 32, 2023.

[76] M. Esposito and D. Falessi, "Uncovering the hidden risks: The importance of predicting bugginess in untouched methods," in *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2023, pp. 277–282.

[77] X. Liu, Y. Zhou, Z. Lu, Y. Mei, Y. Yang, J. Qian, and Y. Zhou, "Unveiling the impact of unchanged modules across versions on the evaluation of within-project defect prediction models," *Journal of Software: Evolution and Process*, vol. 36, no. 12, p. e2715, 2024.

[78] Q. Song, Y. Guo, and M. Shepperd, "A comprehensive investigation of the role of imbalanced learning for software defect prediction," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1253–1269, 2018.

[79] J. Yao and M. Shepperd, "Assessing software defection prediction performance: Why using the matthews correlation coefficient matters," in *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, 2020, pp. 120–129.

[80] C. Tomani, F. K. Waseda, Y. Shen, and D. Cremers, "Beyond in-domain scenarios: robust density-aware calibration," in *International Conference on Machine Learning*. PMLR, 2023, pp. 34 344–34 368.

[81] U. Singhal, R. Feng, X. Y. Stella, and A. Prakash, "Foundation vision models are unsupervised image canonicalizers."