Research Article

An Empirical Study of Software Metrics Diversity for Cross-Project Defect Prediction

Yiwen Zhong,1 Kun Song1,2, ShengKai Lv,1 and Peng He1,2,3

1School of Computer Science and Information Engineering, Hubei University, Wuhan 430062, China
2Hubei Key Laboratory of Applied Mathematics, Hubei University, Wuhan 430062, China
3Hubei Engineering Research Center for Educational Informationization, Wuhan 430062, China

Correspondence should be addressed to Peng He; penghe@hubu.edu.cn

Received 4 September 2021; Revised 24 October 2021; Accepted 5 November 2021; Published 28 November 2021

Abstract

Cross-project defect prediction (CPDP) is a mainsteam method estimating the most defect-prone components of software with limited historical data. Several studies investigate how software metrics are used and how modeling techniques influence prediction performance. However, the software’s metrics diversity impact on the predictor remains unclear. This paper aims to assess the impact of various metric sets on CPDP and investigate the feasibility of CPDP with hybrid metrics. Based on four software metrics types, we investigate the impact of various metric sets on CPDP in terms of $F$-measure and statistical methods. Then, we validate the dominant performance of CPDP with hybrid metrics. Finally, we further verify the CPDP-OSS feasibility built with three types of metrics (orient-object, semantic, and structural metrics) and challenge them against two current models. The experimental results suggest that the impact of different metric sets on the performance of CPDP is significantly distinct, with semantic and structural metrics performing better. Additionally, trials indicate that it is helpful for CPDP to increase the software’s metrics diversity appropriately, as the CPDP-OSS improvement is up to 53.8%. Finally, compared with two baseline methods, TCA+ and TDSelector, the optimized CPDP model is viable in practice, and the improvement rate is up to 50.6% and 25.7%, respectively.

1. Introduction

In software engineering, the conventional defect prediction approach trains a predictor using historical data of the target project and then uses it to predict defects in the subsequent version or release. This process is named as within-project defect prediction (WPDP). However, the cold-start problem is fatal for some new projects or inactive WPDP software projects. Hence, cross-project defect prediction (CPDP) overcomes this issue and has attracted much attention in recent years. In general, CPDP refers to predicting defects in a project using a predictor trained on historical data of other projects [1–3].

As illustrated in Figure 1, various software metrics such as static code, process, object-oriented, and network metrics have been employed for defect prediction. Several studies have also confirmed the discrepancy in the performance of WPDP with different metric sets [4, 5]. For example, Radjenovic et al. [4] highlight that object-oriented and process metrics perform better among six categories of software metrics.

As an artifact, the software can also be abstracted into a coarse-grained network structure based on the dependencies between classes, namely, a class dependency network (CDN) scheme. In CDN, each class is considered a node, and the dependencies between classes are the directed edges. From the perspective of complex networks, some researchers have verified that network metrics are better than code metrics in defect prediction [6, 7]. Additionally, deep learning for network analysis and specifically network embedding learning of a graph structure has attracted significant attention. This strategy aims to find a mapping function to transform each node into a low-dimensional representation. Such techniques involve deepwalk [8], node2vec [9], and...
struc2vec [10]. Qu et al. [11] automatically learn the node representation from the CDN using network embedding for software defect prediction and achieve quite appealing results.

Recently, some researchers highlight that in addition to the features represented by a series of handcraft metrics, software programs have well-defined syntax, represented by abstract syntax trees (ASTs), and rich semantic information hidden deep in the source code. The work of [12–14] has already demonstrated that programs’ semantic information helps characterize defects and improve defect prediction. Specifically, the semantic representation automatically learns from source code, exploiting a deep learning model that distinguishes code regions of different semantics.

Although some studies have investigated the possible benefits of including some measures such as static code and process metrics, none systematically assesses the impact of using various sets of metrics on defect prediction, especially CPDP. The information presented by different software metric sets commonly exhibits significant differences, especially in the context of cross-project. In other words, whether different software metric sets have a significantly distinct effect on the CPDP performance is still an open problem.

This paper focuses on comparative analysis and assesses the impact of using different metric sets to mitigate such problems. Furthermore, it also explores the optimal combination of various metrics in CPDP. The main contributions of this paper are summarized as follows:

1. We conduct a series of experiments and verify that the impact of different types of software metric sets on the performance of CPDP is significantly distinct, with the semantic metric being the most appealing, followed by the structural metric

2. We find that the predictor built with the combination OSS (CK-OO, semantic, and structural metrics) performs best and achieves better performance than several state-of-the-art methods in terms of the F-measure metric

The remainder of this paper is organized as follows. Section 2 reviews the related work in CPDP. Sections 3 and 4 describe the approach of our empirical study and the detailed experimental setups, respectively, while Sections 5 and 6 analyze and discuss the experimental results. Section 7 presents some threats to validity that may affect our study. Finally, Section 8 concludes this paper and presents the directions for future work.

2. Related Work

2.1. Cross-Project Defect Prediction. In recent years, the topic of CPDP has attracted considerable attention from both academia and industry. The most fundamental issues are how to pick the appropriate source projects for a target project and how to train a more accurate predictor through various strategies.

Turhan et al. [2] first utilize the nearest-neighbor filtering technique to prune irrelevant cross-project data, while Porto et al. [15] propose an instance filtering method by selecting the most similar instances from the training data set. Ryu et al. [16] suggest a method of hybrid instance selection using the nearest neighbor (HISNN). The results highlight that instances with solid local knowledge can be identified utilizing nearest-neighbors with the same class label.

To improve the performance of CPDP, Ni et al. [17] develop the FeSCH method and design three ranking strategies to choose the appropriate features. He et al. [18] study CPDP from the perspective of feature simplification.
and compare the performance between CPDP and WPDP. Li et al. [19] compare some well-known data filters and propose an HSBF (hierarchical select-based filter) method. Li et al. [20] analyze the impact of selection granularity of the training data on CPDP and propose a multigranularity selection strategy.

Additionally, Zhang et al. [21] provide an unsupervised approach entitled MT+ to determine the most suitable source project for each target project by considering the impact of various data transformations on the CPDP model. Kumar et al. [22] built a transfer learning scheme for CPDP by utilizing machine learning and identifying the best training data combination. Ryu et al. [23] develop a transfer cost-sensitive boosting method by considering distributional characteristics and the data imbalance for CPDP. They also [24] propose a multiobjective Naive Bayes learning method for CPDP by considering the class imbalance contexts. Poon et al. [25] suggest a credible theory-based Naive Bayes (CNB) classifier and establish a reweighting mechanism for CPDP between the source and target projects.

Besides, to address the heterogeneous defect data sets, He et al. [26] introduce a CPDP-IFS approach based on the distribution characteristics of both the source and target projects. Nam et al. [27] suggest an improved method, HDP, where the metric selection and matching build a defect predictor. Jing et al. [28] propose a unified metric representation for heterogeneous defect data named UMR. Yu et al. [29] present a feature matching and transfer (FMT) approach. Muddu et al. [30] tested the robustness of CPDP experimental research.

Considering CPDP, Herbold et al. [31] replicate 24 approaches proposed between 2008 and 2015 and evaluate their performance on five data sets. The authors claim that CPDP’s model performance is sufficient for practical applications. Goel et al. [32] summarize independent variables, modeling techniques, performance evaluation criteria, and different approaches in building CPDP models but lack a more in-depth impact analysis.

With the extensive application of deep learning technology in various fields, its powerful feature generation ability has also been used for defect prediction [11–14]. For example, Wang et al. [13] generate the source code ASTs and automatically learn the program’s hidden semantic and syntax features through a deep belief network. Li et al. [14] extract the structural information from ASTs through CNN and combine the semantic features with standard code features to improve the performance of software defect prediction. However, ASTs encapsulate only the abstract syntax structure of the source code, which cannot represent the program’s execution process. Phan et al. [33] propose transforming the source code into program control flow graphs (CFG) to extract deeper semantic features from the code. Qu et al. [11] leverage a network embedding technique to automatically learn to encode the program’s class dependency network structure into low-dimensional vector spaces to improve software defect prediction.

2.2. Software Metric. Software quality improvement through defect prediction has been relying on a wide variety of software metrics treated as features. To comprehend the relationship between diverse software metric sets for defect prediction, Chamoli et al. [34] analyze the performance of prediction models based on various software metrics and conclude that software metrics may indeed affect the models’ defect prediction accuracy.

Madeyski et al. [35] identify that process metrics are worth collecting and improve the metric-based prediction models when data sets are collected from a wide range of software projects. Han et al. [36] combine code and process metrics as features and confirm that the predictive capabilities of using two features (BD max and Pre-defects) are comparable to the results of using all 61 features. Öztürk et al. [37] suggest that quality metrics are superior in predicting imbalanced data sets than static code metrics. Xia et al. [38] search for the most critical software metrics and conclude that fewer than 10 metrics can better perform than utilizing 22 or more metrics.

Bluemke et al. [39] describe the process of choosing appropriate metrics for defect prediction. Accordingly, Jiarpakdee et al. [40] suggest that researchers should be aware of redundant metrics before constructing a defect prediction model to maximize their studies’ internal validity. Caglayan et al. [41] conclude that the performance of different metric sets in building a defect prediction model depends on the project’s characteristics and the targeted prediction level.

Mauśa et al. [42] replicate the case study of deriving thresholds for software metrics using a statistical model based on logistic regression and analyze a more comprehensive set of software metrics. The results reveal that the threshold values of some metrics can be used to predict defect-prone modules effectively. Recently, Zhang et al. [43] suggest that an aggregation scheme can significantly alter correlations among metrics and correlations between metrics and the defect count through an analysis of 11 aggregation schemes using data collected from 255 open-source projects.

3. Problem and Method

3.1. Research Question. This paper defines CPDP as follows: given a source project $P_s$ and a target project $P_t$, CPDP aims to achieve the target prediction in $P_t$ using the knowledge extracted from $P_s$, where $P_s \neq P_t$. Let the source and target projects share the same feature cardinality and metric sets. The goal of CPDP is to learn a model from the selected source projects (training data) and apply it to the target project (test data). In the context examined here, project $P_s$ as a defect data set, contains $m$ instances represented as $P_s = \{I_{j1}, I_{j2}, \ldots, I_{jm}\}$. An instance is $I_i = \{f_{i1}, f_{i2}, \ldots, f_{id}, L\}$, where $f_{ki}$ is the $k$–th dimension of the representation vector of instance $I_i$, and $d$ denotes the total dimensions, namely, the scale of the metrics. If instance $I_i$ is buggy, then $L$ is one; otherwise, $L$ is zero.
As mentioned in Section 2.2, various metrics can measure software complexity and quality in practice; therefore, a defect data set may contain multiple types of software metrics. According to the statistics, most public defect data sets contain at least two types of software metrics. For example, the commonly used AEEEM data set involves two sets of software metrics. According to the existing practice [11–13], deep learning technology can provide structural and semantic metrics. In other words, there are at least four sets of metrics.

Nevertheless, only a few research works based on these data sets have explored the impact of different metric sets and their combinations on the performance of CPDP, especially for handcrafted and automatically learned metrics. Spurred by that, this paper aims to find empirical evidence addressing the following three research questions.

RQ1: Is the impact of different metric sets on the performance of CPDP significantly distinct?
RQ2: Does CPDP based on hybrid metrics perform better?
RQ3: Does the optimized CPDP model outperform the baselines?

3.2. Approach. An effective prediction model affords more resources to be devoted to the bug-prone instances and consequently improves the quality of the latter instances. Existing CPDP models usually aim to improve the learning algorithm and make the predictor perform better. Hence, they always ignore the impact of software metrics on prediction performance. To answer the above research questions, we will consider constructing a CPDP model for two scenarios (Figure 2).

Scenario 1 considers CPDP modeling utilizing a single metric set, which is the most straightforward modeling method. For this case, we will investigate the performance of the predictor. Scenario 2 considers constructing a CPDP model based on multitype metric sets. Note that different colors in Figure 2 distinguish the types of indicator sets. The details of this scenario are provided in the following steps.

Step 1: Defect data sets are constructed for each project according to the software metric types. An instance is described as $I = \{f_{11}, f_{12}, \ldots, f_{1d_1}, \ldots, f_{n1}, f_{n2}, \ldots, f_{nd_n}, L\}$, where $n$ is the software metric category cardinality ($n \geq 1$). As mentioned in Section 3.1, $d_n$ denotes the dimension of the $n$-th set of software metrics, $f$ is the metric value, and $L$ the label.

Step 2: After collecting the defect data sets, we further determine a series of classification algorithms employed to learn the predictor, for example, Naive Bayes, logistic regression, and J48.

Step 3: The corresponding CPDP model based on the selected defect data sets and classification algorithms is constructed according to the specific scenario requirement.

4. Experimental Setup

4.1. Data Sets. We investigate our study on the public AEEEM data set [44], which involves five open-source projects. Table 1 lists the details of the five projects, where the second and third columns are the number of the defective and the clean instances, respectively. Each project refers to process and CK-OO metrics. Each instance represents a class file and comprises software metrics and a dependent variable labeling defining if bugs exist in this class file. Table 2 presents all metric sets involved in our study.

Note that we expand the set of metrics to the existing data set, including structural and semantic metrics. The former is extracted from a class dependency network employing network embedding learning. Specifically, this paper utilizes the node2vec method [9] to map each class node to a low-dimensional vector. Regarding the semantic metrics, these adopt the method of [13]. Among the above metrics, the traditional code metrics are not listed because they have little effect on CPDP and are not applicable to the current data set.

Data imbalance is a crucial and unavoidable problem in software defect prediction. In our data set, due to the absence of defects, the number of nonbuggy samples is far more than the number of defective samples, with the imbalanced distribution seriously affecting the prediction accuracy. To overcome this problem, we balance the data sets with SMOTE. Additionally, since the scale of the numerical metric values is different in a data set, we normalize the metric values within the range of [0, 1] utilizing the z-score method.
4.2. Experimental Design. This section describes the entire experimental framework according to the previous three research questions, as illustrated in Figure 3.

First, to conduct an impact analysis among all four metric sets, the CPDP experiments are conducted in the first scenario. This trial analyzes the differences of the software metric sets under a specific classifier. Then, we expand the experiments in the following scenario and compare the average prediction results of six cases involving different combination patterns. Finally, based on the optimal metric combination, we further verify the feasibility of the proposed CPDP model by challenging it against two current models.

Once this process is completed, the answers to the three research questions of our study will be discussed.

4.3. Classifiers. Machine learning algorithms are widely used in defect prediction, with the classification algorithms being able to classify the defective modules correctly. This paper utilizes four typical classification algorithms as the primary learning algorithms.

(i) Logistic regression (LR): A widely used supervised classification algorithm that essentially solves a dichotomy problem. Due to its universality and practicability, several methods employ it for defect prediction [5, 18, 26].

(ii) Random forest (RF): A classifier that uses multiple trees to train and predict samples, aiming at reducing variance. RF has a better generalization and classification capability than typical decision trees [18].

(iii) Naïve Bayes (NB): The simplest classifier based on Bayesian theory and independent hypothetical testing. It is widely accepted that NB outperforms other classifiers and thus is frequently used to build defect prediction models [5, 18, 23].

(iv) J48: A high-efficiency decision tree algorithm that uses the greedy technique for supervised classification, posing an appealing tool for defect prediction [18].

4.4. Evaluation Measures. To predict whether an instance (class file) is defective, we use binary classification technology. The possible results are true positive (TP), false positive (FP), false negative (FN), and true negative (TN). The conventional classification evaluation measures include

| Table 1: AEEEM data set. |
|--------------------------|
| Project | Number of buggy instances | Number of non-buggy instances | The total number of instances | Buggy ratio (%) |
| Eclipse | 206 | 792 | 998 | 20.64 |
| Equinox | 129 | 196 | 325 | 39.69 |
| Lucene | 64 | 627 | 691 | 9.26 |
| Mylyn | 245 | 1,617 | 1,862 | 13.16 |
| Pde | 209 | 1,288 | 1,497 | 13.96 |

| Table 2: Four types of software metrics will be involved in this paper. |
|--------------------------|
| Category | Metric description |
| Process metrics (15-dimension) | numberOfVersionsUntil | avgLinesAddedUntil |
| | numberOfFixesUntil | linesRemovedUntil |
| | numberOfRefactoringsUntil | maxLinesAddedUntil |
| | numberOfAuthorsUntil | maxLinesRemovedUntil |
| | linesAddedUntil | avgLinesRemovedUntil |
| | codeChurnUntil | ageWithRespectTo |
| | weightedAgeWithRespectTo |
| CK-OO metrics (17-dimension) | Coupling between object classes |
| | Depth of Inheritance Tree |
| | Afferent Couplings |
| | Efferent Couplings |
| | Lack of Cohesion in Methods |
| | Number of Children |
| Structural metrics | A d-dimensional space of features (d = 32) |
| Semantic metrics | A d-dimensional space of features (d = 32) |

Figure 3: The experimental framework according to the three research questions.
precision, recall, and $F$-measure expressed as follows. Given the contradiction between precision and recall, we use $F$-measure to evaluate the prediction performance.

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

$$F \text{- measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$  \hspace{1cm} (1)$$

Additionally, statistical tests assist in understanding whether a statistically significant difference between two results exists. This work utilizes the Wilcoxon signed-rank test to check whether the performance difference between the prediction models with different software metrics is significant. To further examine the effectiveness and following the work of [13, 18, 26], we employ Cliff’s delta ($\delta$) to measure the effect size of our approach. Cliff’s delta is a nonparametric effect size measurement scheme that quantifies the difference between the two approaches. Table 3 describes the meanings of various $\delta$ values [45].

### 5. Experimental Results

This section reports the experimental results aiming at answering the three research questions formulated in Section 3.1.

**RQ1:** Is the impact of different metric sets on the performance of CPDP significantly distinct?

This trial considers Scenario 1, with the corresponding results presented in Figure 4, and highlights that the $F$-measure values obtained using different metric sets are generally different, implying different levels of influence on CPDP. On the one hand, the prediction results of CPDP models based on different metric sets vary when the classifier is consistent. For instance, in Figure 4(a) and considering the semantic and process metrics, the medium values of the $F$-measure are 0.381 and 0.334, respectively, inferring that the semantic metric performs much better than the process metric under the J48 classifier. Additionally, a value of 0.381 is a relatively high index showing that the semantic metrics perform better than other metrics, whose index values are 0.334, 0.337, and 0.343.

Under different classifiers, the advantages of specific metrics are also unstable. Considering the semantic metric as an example, Figure 4(a) indicates that this metric performs best (0.381), but in Figures 4(c) and 4(d), this advantage is less obvious. Note that compared with the semantic metrics, other metrics perform the same instability (approximately 0.3–0.4), sometimes leading to more outliers.

Note: A negative $\delta$ value represents that the result of the latter metric set is better; on the contrary, the former is better.

To further distinguish the impact of different metric sets on CPDP, we evaluate the results in terms of the Wilcoxon signed-rank test ($p$ value) and Cliff’s delta ($\delta$) metric. In this study, we statistically analyze four types of metrics based on the null hypothesis, that is, two metric sets have the distribution of the same results. In Table 4, the Wilcoxon signed-rank test highlights no significant difference between the semantic and structural metrics and between process and CK-OO metrics, as both $p$ values exceed 0.05. However, the differences between the two groups are statistically significant, especially between the CK-OO and semantic metrics ($p$ value = 0.003).

In Table 4, the effect size $\delta$ between the structural and semantic metrics is small, and the metric is minimal between the CK-OO and process metrics. Considering the process and CK-OO metrics, the dominant effect size of the semantic metrics tends to be larger due to the negative $\delta$ values ($-0.445$ and $-0.408$), while the dominant effect size of structural metrics tends to be medium ($\delta = 0.295$).

Therefore, overall, the semantic metric performs best followed by the structural metric and then the CK-OO and process metrics.

In conclusion, based on the experimental results, the impact of different metric sets on the performance of CPDP is distinct, with a significant difference.

**RQ2:** Does CPDP based on hybrid metrics perform better?

To answer this research question, we construct a defect predictor using the logistic regression as described in Scenario 2. To simplify the presentation, we label the CPDP model with the Process and Semantic metric as CPDP-PS, OS (CK-OO and Semantic), SS (Structural and Semantic), OSS (CK-OO, Structural, and Semantic), and POSS (Process, CK-OO, Structural, and Semantic). Table 5 presents the prediction results of each target project in terms of $F$-measure values. The results indicate that CPDP-OSS performs best due to the greater $F$-measure values. For example, for Eclipse, the $F$-measure value of CPDP-OSS is 0.536. Compared to the remaining combinations, the performance is higher by 29.22%, 19.73%, 4.93%, and 9.58%, respectively. Additionally, the improvement increases to 62.62% compared to the previous CPDP involving single semantic metrics.

Additionally, for Equinox, the performance of CPDP-OSS exceeds 0.6 when using Eclipse, Lucene, and Pde as the source project. Note that compared with CPDP-PS and CPDP-OS, the performance improvement of CPDP-OSS for Mylyn is more prominent, exceeding 30%. Interestingly, for Lucene and Pde, their $F$-measure values of CPDP with single semantic metrics are more significant than that of CPDP.

| Effect size  | $|\delta|$ |
|--------------|---------|
| Very small   | 0.008   |
| Small        | 0.147   |
| Medium       | 0.33    |
| Large        | 0.474   |
| Very large   | 0.622   |
| Huge         | 0.811   |

**Table 3:** The mappings between different $\delta$ values and their effective levels.
Figure 4: The standardized boxplots of the performance of CPDP are achieved by different predictors based on different types of software metrics and classifiers in Scenario 1. From the bottom to the top of a standardized box plot: minimum, first quartile, median, third quartile, and maximum. Any data not included between the whiskers are plotted as a small circle. (a) J48, (b) LR, (c) NB, and (d) RF.

Table 4: The results of the Wilcoxon signed-rank test and Cliff’s effect size ($\delta$).

|                     | Process vs. semantic | Structure vs. semantic | CK-OO vs. semantic | Structure vs. process | CK-OO vs. process | CK-OO vs. semantic |
|---------------------|----------------------|------------------------|--------------------|-----------------------|-------------------|--------------------|
| Sig. $p$ value (0.05) | 0.022                | 0.49                   | 0.003              | 0.024                 | 0.5               | 0.018              |
| $\delta$ value      | -0.445               | -0.083                 | -0.408             | 0.295                 | 0.00              | -0.295             |

Table 5: Comparison of the CPDP performance in terms of $F$-measure.

| Metrics   | Eclipse | Equinox | Target | Mylyn | Lucene |  |
|-----------|---------|---------|--------|-------|--------|---|
| Semantic  | 0.330 (+62.62%) | 0.385 (+55.93%) | 0.450 (+0.22%) | 0.442 (+12.06%) | 0.454 (+4.01%) | |
| PS        | 0.415 (+29.22%) | 0.528 (+13.63%) | 0.382 (+18.06%) | 0.379 (+30.61%) | 0.380 (+24.21%) | |
| OS        | 0.448 (+19.73%) | 0.566 (+6.17%) | 0.408 (+10.54%) | 0.370 (+33.78%) | 0.413 (+14.29%) | |
| SS        | 0.511 (+4.93%) | 0.584 (+2.83%) | 0.451 (—) | 0.479 (+3.34%) | 0.453 (+4.19%) | |
| OSS       | 0.536 | 0.601 | 0.451 | 0.495 | 0.472 | |
| POSS      | 0.489 (+9.58%) | 0.583 (+3.07%) | 0.424 (+6.37%) | 0.472 (+4.87%) | 0.436 (+8.26%) | |

Note: The percentages within the parentheses are the improvement rates of CPDP-OSS, and the boldfaced numbers represent the best results in the same context.
with hybrid metrics, except for that of CPDP-OSS. Besides, the performance of CPDP-SS is very close to that of CPDP-OSS, even on Lucene, as the $F$-measure values are the same. Therefore, it can be concluded that sometimes "more is not better."

Overall, the results indicate that CPDP with CK-OO, structural, and semantic metrics can identify more buggy instances than the other combinations examined. Therefore, it is mandatory to consider the effect of hybrid metrics on CPDP.

RQ3: Does the optimized CPDP model outperform the baselines?

The previous results validate that it is still valuable to consider hybrid metrics during CPDP modeling. To evaluate the practicability and usefulness of CPDP-OSS, we built CPDP models using two existing approaches, that is, TCA+ [46] and TDSelector [47], and perform experiments on the same data set. Table 6 presents the comparative results between our approach and the two baselines, where the max $F$-measure value per row is in bold. CPDP-OSS outperforms both baselines, as most boldfaced $F$-measure values, that is, 12 out of 20, and the average improvement rates of $F$-measure values belong to CPDP-OSS.

Compared with TCA+, Table 6 highlights that 6 out of 20 improved rates of our approach exceed 20%, while the maximum is 50.6%. Regarding TDSelector, four cases pose an improvement exceeding 20%, and the maximum is 25.7%. With this evidence, the proposed CPDP-OSS approach is validated to be beneficial for improving the performance of a CPDP model.

6. Discussion

RQ1: Our experimental results suggest that the impact of various metrics sets on the performance of CPDP is distinct in terms of $F$-measure. Our findings indicate that the semantic metrics, on average, yield the best CPDP models, with structural metrics to follow. Meanwhile, according to Table 2, it is evident that the scale of these two metrics sets in our defect data sets is more significant, that is, a more extensive set of metrics may lead to better prediction. Thus, assisted by deep learning technology, deeper information can be automatically learned from the program.

The CK-OO metric and the process metrics are the most frequently used for defect prediction, with the authors in [4] arguing that the CK-OO metric has a good explanation and predictive power. Nevertheless, in our trials, both do not perform so well as expected. A possible explanation is a difference in the prediction context, leading to the conclusion that cross-project defect prediction is different from the traditional within-project defect prediction.

RQ2: For CPDP, the effectiveness of increasing the metrics diversity is proven, broadly consistent with some prior studies’ findings. Considering software metrics, D’Ambros et al. [43] investigate the prediction based on a single set of metrics and found that defect prediction models based on a single set of metrics are unstable. Hall et al. [5] also found that defect prediction models using a comprehensive combination of metrics perform well.

According to the experimental results, overall, using semantic and structural indicators affords a good prediction capability. One possible explanation is that when using AST to extract semantic information from code, the complexity of the source code has been achieved to a certain extent. Therefore, when the CK-OO metric is continuously considered, the improvement is limited.

RQ3: In the proposed CPDP-OSS approach, the advantage relies on the implicit diversity among software metrics. In Table 7, although the results show an overall improvement in the predictive performance of CPDP-OSS, the advantage is not apparent due to the $p$ values exceeding 0.05 and the small effectiveness levels.

Several factors may prohibit revealing the apparent advantage of the proposed approach. Additionally, due to the limitation of the data set, only three/four types of software metrics are introduced, and we employ the most basic semantic and structural metrics learning model. Currently, some improved deep learning models have been used to solve this task.

To ease the complexity of the proposed approach, instead of using the complex and representative boosting and bagging algorithms, we utilize a simple logistic regression. Therefore, there is much room for improving our approach, and we believe that the advantages will be more evident after some improvement.

Although the advantages of the CPDP-OSS approach are not particularly obvious, it is more efficient than the two baseline competitor methods. Regarding TDSelector, for each project, it requires manually calculating 76 metrics. However, after applying the classification on the semantic and structural features, we can avoid cumbersome calculations through machine learning and then reduce the metrics calculation to $17 + 15 + 2 = 34$, which significantly improves data processing efficiency. From this point of view, our experimental results still have great application value.

These two baselines are currently the most in-depth and representative baselines in our experimental research. There may be better baselines for comparison, and in future work, we will continue to follow up and compare them.

7. Threats to Validity

From this work, several meaningful results are obtained, but potential threats to the validity of our work remain.

Threats to construct validity primarily regard the software metrics used in this paper. The experimental data set employed from [42] is a public defect data set. According to the authors’ statement, inevitably, some links between the bug database and the source code repositories are missing. However, these data have been applied to numerous prior studies, and therefore, we argue that our results are credible and representative.

Threats to internal validity concern any confounding factor that may affect our results. First, we adopt the commonly used SMOTE method to preprocess the defect data sets due to the imbalanced data. As far as we know,
SMOTE-based oversampling techniques were widely adopted as the selection to handle the class imbalance problem in software defect prediction [48–51]. Although many improved sampling techniques have been proposed, it is reasonable to believe that it is feasible to use SMOTE-based oversampling technology in this paper. Second, any feature selection method is not introduced during the CPDP modeling, and third, a simple connection method is directly used to generate the hybrid metrics in RQ2. Undoubtedly a complex fusion mechanism will result in better performance and greater calculation time. Fourth, we train the predictors for each classifier based on the default parameter settings configured by the Weka API. Hence, we are indeed aware that the results of our study would change if we use different settings of the above four factors.

Threats to statistical conclusion validity concern the relationship between the treatment and the outcome. In addition to the intuitive comparison of the prediction results in terms of $F$-measure, this paper also utilizes the Wilcoxon signed-rank test and Cliff’s delta effect size to compare the results. According to the significance criteria and effectiveness levels, the results indicate that the difference of various software metrics is distinct, and the introduction of diversity among software metrics is valuable. However, the advantage of our method is not noticeable compared to the two baseline methods, indicated by $|\delta|$ that is approximately 0.12.

Threats to external validity emphasize the generalization of the findings. Predictions in this paper are constructed on five open-source software systems. Although our experiments can be repeated with more open-source projects and developed with different software metrics, the empirical results for industrial software projects may differ from our main conclusions. We minimize this threat by selecting a data set that consists of parts of Eclipse, an open-source system with a solid industrial background.

8. Conclusions

This paper reports a comparative study of software metrics selection for CPDP, aiming to maximize the CPDP model’s diversity in terms of metric sets. Four types of software metrics are considered for modeling, and a series of experiments are conducted on five open-source projects. The study consists of (1) the impact analysis of different metric sets on CPDP, (2) exploration of the metrics’ combination, and (3) comparison between CPDP built with hybrid metric sets (CPDP-OSS) and two current state-of-the-art approaches.

The results indicate that the impact of different metric sets on CPDP is significantly distinct. Additionally, our trials indicate it is helpful for CPDP to increase the diversity of software metrics appropriately, and there are significant improvements between CPDP-OSS and the remaining combinations examined. The most significant improvement rate is up to 53.8%. Our results also highlight that CPDP-OSS outperforms two benchmarks, and the most considerable improvement rate is up to 50.6% and 25.7%.

### Table 6: Comparison between the proposed approach and two baseline methods: TCA+ and TDSelector.

| Source/target            | TCA+       | TDSelector | CPDP-OSS  |
|--------------------------|------------|------------|-----------|
| Eclipse/Equinox          | 0.600 (+5.9%) | 0.514 (+23.7%) | 0.636     |
| Lucene/Equinox           | 0.620 (+4.4%) | 0.664 (-2.5%) | 0.648     |
| Mylyn/Equinox            | **0.560 (-11.6%)** | 0.459 (+7.8%) | 0.495     |
| Pde/Equinox              | 0.600 (+4.0%) | 0.610 (+2.3%) | **0.624** |
| Equinox/Eclipse          | **0.540 (-4.5%)** | 0.519 (-0.7%) | 0.516     |
| Lucene/Eclipse           | 0.560 (-4.5%) | **0.594 (-9.9%)** | 0.535     |
| Pde/Eclipse              | 0.430 (+27.7%) | 0.455 (+20.7%) | **0.549** |
| Mylyn/Eclipse            | 0.480 (+13.5%) | 0.510 (+5.9%) | **0.545** |
| Equinox/Lucene           | 0.270 (+50.6%) | 0.337 (+20.7%) | **0.407** |
| Eclipse/Lucene           | 0.310 (+26.9%) | 0.313 (+25.7%) | **0.393** |
| Mylyn/Lucene             | 0.250 (+32.0%) | 0.298 (+10.7%) | **0.330** |
| Pde/Lucene               | **0.330 (-4.9%)** | 0.314 (+0.0%) | 0.314     |
| Equinox/Mylyn            | 0.230 (+24.7%) | 0.264 (+8.6%) | **0.287** |
| Eclipse/Mylyn            | **0.360 (-13.1%)** | 0.303 (+3.2%) | 0.313     |
| Lucene/Mylyn             | 0.290 (+6.4%) | 0.291 (+6.1%) | **0.309** |
| Pde/Mylyn                | 0.290 (-6.1%) | 0.300 (-9.2%) | 0.272     |
| Equinox/Pde              | 0.330 (+21.2%) | 0.367 (+9.0%) | **0.400** |
| Eclipse/Pde              | 0.380 (+2.8%) | 0.391 (+0.0%) | **0.391** |
| Lucene/Pde               | **0.370 (-11.8%)** | 0.365 (-10.6%) | 0.326     |
| Mylyn/Pde                | 0.370 (+0.4%) | 0.371 (+0.0%) | **0.371** |
| Average                  | 0.410 (+6.0%) | 0.412 (+5.1%) | **0.433** |

### Table 7: The results of the Wilcoxon signed-rank test and Cliff’s effect size ($\delta$).

|                  | CPDP-OSS vs. TCA+ | CPDP-OSS vs. TDSelector |
|------------------|-------------------|-------------------------|
| Sig. p value     | 0.189             | 0.096                   |
| $\delta$ value   | 0.129             | 0.134                   |
respectively. Therefore, it is meaningful to introduce the diversity of metric sets to improve the performance of CPDP.

Future work shall mainly focus on collecting more open-source projects to validate our approach’s generalization and improve the learning techniques of code semantic and structural information to provide a more effective CPDP model for defect prediction. The results of the \( p \) values and the small Cliff’s delta values in the experiment show that compared with the two baselines, the effect of CPDP-OS is not very significant. We will make improvements in future work and continue to experiment and test.

**Data Availability**

We investigate our study on the public AEEEM data set [44], which involves five open-source projects. Each project refers to process and CK-OO metrics. Each instance represents a class file and comprises software metrics and a dependent variable labeling defining if bugs exist in this class file.

**Conflicts of Interest**

The authors declare that they have no conflicts of interest.

**Acknowledgments**

The authors greatly appreciate Dr. Jaechang Nam and Dr. Sinno Jialin Pan, the authors of reference [46], for providing us with the TCA source program and friendly teaching us how to use it. This work was supported by the National Key R&D Program of China (No. 2018YFB1003801), the National Natural Science Foundation of China (Nos. 61832014 and 61902114), the Science and Technology Innovation Program of Hubei Province under Grant (Nos. 2018ACA133 and 61902114), and the Open Foundation of Hubei Key Laboratory of Applied Mathematics (No. HBAM201901).

**References**

[1] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, “An investigation on the feasibility of cross-project defect prediction,” *Automation of Software Engineering*, vol. 19, no. 2, pp. 167–199, 2012.

[2] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, “On the relative value of cross-company and within-company data for defect prediction,” *Empirical Software Engineering*, vol. 14, no. 5, pp. 540–578, 2009.

[3] Y. Ma, G. Luo, X. Zeng, and A. Chen, “Transfer learning for cross-company software defect prediction,” *Information and Software Technology*, vol. 54, no. 3, pp. 248–256, 2012.

[4] D. Radjenović, M. Heričko, R. Torkar, and A. Živković, “Software fault prediction metrics: a systematic literature review,” *Information and Software Technology*, vol. 55, no. 8, pp. 1397–1418, 2013.

[5] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, “A systematic literature review on fault prediction performance in software engineering,” *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276–1304, 2012.

[6] Y. Qu, Q. Zheng, J. Chi et al., “Using K-core decomposition on class dependency networks to improve bug prediction model’s practical performance,” *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 348–366, 2021.

[7] G. Concas, M. Marchesi, C. Monni, M. Orrù, and R. Tonelli, “Software quality and community structure in java software networks,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 27, no. 7, pp. 1063–1096, 2017.

[8] B. Perozzi, R. Al-Rfou, and S. Skiena, “Deepwalk: online learning of social representations,” in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA, August 2014.

[9] A. Grover and J. Leskovec, “node2vec: scalable feature learning for networks,” in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. San Francisco, CA, USA, August 2016.

[10] L. F. R. Ribeiro, P. H. P. Saverese, and D. R. Figueiredo, “struc2vec: learning node representations from structural identity,” in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Halifax, NS, Canada, August 2017.

[11] Y. Qu, T. Liu, J. Chi et al., “node2defect: using network embedding to improve software defect prediction,” in *Proceedings of the 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Montpellier, France, September 2018.

[12] Z. Cai, L. Lu, and S. Qiu, “An abstract syntax tree encoding method for cross-project defect prediction,” *IEEE Access*, vol. 7, pp. 170844–170853, 2019.

[13] S. Wang, T. Liu, J. Nam, and L. Tan, “Deep semantic feature learning for software defect prediction,” *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1267–1293, 2018.

[14] J. Li, P. He, J. Zhu, and M. R. Lyu, “Software defect prediction via convolutional neural network,” in *Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, Prague, Czech Republic, July 2017.

[15] F. Porto and A. D. S. Simao, “Feature subset selection and instance filtering for cross-project defect prediction - classification and ranking,” *CLEI Electronic Journal*, vol. 19, no. 3, pp. 4–17, 2016.

[16] D. Ryu, J.-I. Jang, and J. Baik, “A hybrid instance selection using nearest-neighbor for cross-project defect prediction,” *Journal of Computer Science and Technology*, vol. 30, no. 5, pp. 969–980, 2015.

[17] C. Ni, W. Liu, Q. Gu, X. Chen, and D. Chen, “FeSCH: a feature selection method using clusters of hybrid-data for cross-project defect prediction,” in *Proceedings of the Computer Software and Applications Conference*, pp. 51–56, IEEE, Turin, Italy, July 2017.

[18] P. He, B. Li, X. Liu, J. Chen, and Y. Ma, “An empirical study on software defect prediction with a simplified metric set,” *Information and Software Technology*, vol. 59, pp. 170–190, 2015.

[19] Y. Li, Z. Huang, Y. Wang, and B. Fang, “Evaluating data filter on cross-project defect prediction: comparison and improvements,” *IEEE Access*, vol. 5, no. 99, pp. 25646–25656, 2017.

[20] Y. Li, P. He, B. Li, and Y. Ma, “Multi-granularity selection of training data for cross-project defect prediction,” *Journal of Chinese Computer Systems*, vol. 38, no. 9, pp. 1934–1939, 2017.

[21] F. Zhang, J. Keivanloo, and Y. Zou, “Data transformation in cross-project defect prediction,” *Empirical Software Engineering*, pp. 1–33, 2017.

[22] K. Lov, R. Santanu, and B. Ranjan, “Transfer learning for cross-project change-proneness prediction in object-oriented software systems,” *ACM SIGSOFT-Software Engineering Notes*, vol. 42, no. 3, pp. 1–11, 2017.
[23] D. Ryu, J. I. Jang, and J. Baik, “A transfer cost-sensitive boosting approach for cross-project defect prediction,” *Software Quality Journal*, vol. 25, no. 1, pp. 1–38, 2017.

[24] D. Ryu and J. Baik, “Effective multi-objective naive Bayes learning for cross-project defect prediction,” *Applied Soft Computing*, vol. 49, pp. 1062–1077, 2016.

[25] W. N. Poon, K. E. Bennim, J. Huang, P. Phannachitta, and J. W. Keung, “Cross-project defect prediction using a credibility theory based naive Bayes classifier,” in *Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, July 2017.

[26] P. He, B. Li, and Y. Ma, “Towards cross-project defect prediction with imbalanced feature sets,” 2014, https://arxiv.org/abs/1411.4228.

[27] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, “Heterogeneous defect prediction,” in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 44, pp. 508–519, 2015.

[28] X. Jing, F. Wu, X. Dong, F. Qi, and B. Xu, “Heterogeneous cross-company defect prediction by unified metric representation and CCA-based transfer learning,” in *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pp. 496–507, Bergamo, Italy, August 2015.

[29] Q. Yu, S. Jiang, and Y. Zhang, “A feature matching and transfer approach for cross-company defect prediction,” *Journal of Systems and Software*, vol. 132, pp. 366–378, 2017.

[30] B. Muddu, A. Asadullah, and V. Bhat, “CPDP: a robust technique for plagiarism detection in source code,” in *Proceedings of the 2013 7th International Workshop on Software Clones (IWSC)*, pp. 39–45, San Francisco, CA, USA, May 2013.

[31] S. Herbold, A. Trautsch, and J. Grabowski, “A comparative study to benchmark cross-project defect prediction approaches,” *IEEE Transactions on Software Engineering*, vol. 44, no. 9, p. 1, 2017.

[32] L. Goel, D. Damodaran, S. K. Khatri, and M. Sharma, “A literature review on cross project defect prediction,” in *Proceedings of the IEEE Uttar Pradesh Section International Conference on Electrical, Computer and Electronics*, pp. 680–685, Mathura, India, October 2018.

[33] A. V. Phan, M. Le Nguyen, and T. B. Lam, “Convolutional neural networks over control flow graphs for software defect prediction,” in *Proceedings of the 2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, IEEE, Boston, MA, USA, November 2017.

[34] S. Chamoli, G. Tenne, and S. Bhattacharyya, “Analysing software metrics for accurate dynamic defect prediction models,” *Indian Journal of Science and Technology*, vol. 8, no. 54, pp. 96–100, 2015.

[35] L. Madeyski and M. Jurczko, “Which process metrics can significantly improve defect prediction models? An empirical study,” *Software Quality Journal*, vol. 23, no. 3, pp. 393–422, 2015.

[36] W. Han, C. H. Lung, and S. A. Ajila, “Empirical investigation of code and process metrics for defect prediction,” in *Proceedings of the IEEE Second International Conference on Multimedia Big Data*, pp. 436–439, IEEE, Taipei, Taiwan, April 2016.

[37] M. M. Öztürk, “Which type of metrics are useful to deal with class imbalance in software defect prediction?” *Information and Software Technology*, vol. 92, pp. 17–29, 2017.

[38] Y. Xia, G. Yan, and Q. Si, “A study on the significance of software metrics in defect prediction,” in *Proceedings of the Sixth International Symposium on Computational Intelligence and Design*, pp. 343–346, IEEE Computer Society, Washington, DC, USA, October 2013.

[39] I. Bluemke and A. StepieÌ„, “Selection of metrics for the defect prediction,” *Dependability Engineering and Complex Systems*, Springer International Publishing, New York, NY, USA, 2016.

[40] J. Jiarpakdee, C. Tangthanthavorn, A. Ihara, and K. Matsumoto, “A study of redundant metrics in defect prediction datasets,” in *Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops*, pp. 51–52, IEEE, Ottawa, Canada, August 2016.

[41] B. Caglayan, B. Turhan, A. Bener, and M. Habayeb, “Merits of organizational metrics in defect prediction: an industrial replication,” in *Proceedings of the IEEE International Conference on Software Engineering*, pp. 89–98, IEEE, Florence, Italy, May 2015.

[42] G. Mauà and T. G. Grbac, “The stability of threshold values for software metrics in software defect prediction,” in *Proceedings of the International Conference on Model and Data Engineering*, pp. 81–95, Barcelona, Spain, October 2017.

[43] F. Zhang, A. E. Hassan, S. McIntosh, and Y. Zou, “The use of summation to aggregate software metrics hinders the performance of defect prediction models,” *IEEE Transactions on Software Engineering*, vol. 43, no. 5, pp. 476–491, 2017.

[44] M. D’Ambros, M. Lanza, and R. Robbes, “An extensive comparison of bug prediction approaches,” in *Proceedings of the 2017 7th IEEE Working Conference on Mining Software Repositories (MSR)*, pp. 31–41, IEEE, Cape Town, South Africa, May 2010.

[45] J. D. Long, D. Feng, and N. Cliff, “Ordinal analysis of behavioral data,” *Handbook of Psychology: Research Methods in Psychology*, vol. 2, pp. 635–661, John Wiley & Sons Inc, Hoboken, NJ, USA, 2003.

[46] J. Nam, S. J. Pan, and S. Kim, “Transfer defect learning,” in *Proceedings of the International Conference on Software Engineering*, pp. 382–391, IEEE, San Francisco, CA, USA, May 2013.

[47] P. He, Y. Ma, and B. Li, “TDSselector: a training data selection method for cross-project defect prediction,” 2016, https://arxiv.org/abs/1612.09065.

[48] S. Feng, J. Keung, X. Yu, Y. Xiao, and M. Zhang, “Investigation on the stability of SMOTE-based oversampling techniques in software defect prediction,” *Information and Software Technology*, vol. 139, no. 6, Article ID 106662, 2021.

[49] C. Pak, T. W. Tian, and H. S. Xiao, “An empirical study on software defect prediction using over-sampling by SMOTE,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 28, no. 6, pp. 811–830, 2018.

[50] Z. Eivazpour and M. R. Keyvanpour, “Improving performance in software defect prediction using variational autoencoder,” in *Proceedings of the 2019 5th Conference on Knowledge Based Engineering and Innovation (KBEI)*, Tehran, Iran, February 2019.

[51] A. Balogun, S. Basri, S. J. Abdulkadir, and V. E. Adeyemo, “Software defect prediction: analysis of class imbalance and performance stability,” *Journal of Engineering Science and Technology*, vol. 14, no. 6, pp. 3294–3308, 2019.