



An Empirical Study on Just-in-time Conformal Defect Prediction

Xhulja Shahini
paluno, University of Duisburg-Essen
Essen, Germany
xhulja.shahini@paluno.uni-due.de

Andreas Metzger
paluno, University of Duisburg-Essen
Essen, Germany
andreas.metzger@paluno.uni-due.de

Klaus Pohl
paluno, University of Duisburg-Essen
Essen, Germany
klaus.pohl@paluno.uni-due.de

ABSTRACT

Code changes can introduce defects that affect software quality and reliability. Just-in-time (JIT) defect prediction techniques provide feedback at check-in time on whether a code change is likely to contain defects. This immediate feedback allows practitioners to make timely decisions regarding potential defects. However, a prediction model may deliver false predictions, that may negatively affect practitioners' decisions. False positive predictions lead to unnecessarily spending resources on investigating clean code changes, while false negative predictions may result in overlooking defective changes. Knowing how uncertain a defect prediction is, would help practitioners to avoid wrong decisions. Previous research in defect prediction explored different approaches to quantify prediction uncertainty for supporting decision-making activities. However, these approaches only offer a heuristic quantification of uncertainty and do not provide guarantees.

In this study, we use conformal prediction (CP) as a rigorous uncertainty quantification approach on top of JIT defect predictors. We assess how often CP can provide guarantees for JIT defect predictions. We also assess how many false JIT defect predictions CP can filter out. We experiment with two state-of-the-art JIT defect prediction techniques (DeepJIT and CC2Vec) and two widely used datasets (Qt and OpenStack).

Our experiments show that CP can ensure correctness with a 95% probability, for only 27% (for DeepJIT) and 9% (for CC2Vec) of the JIT defect predictions. Additionally, our experiments indicate that CP might be a valuable technique for filtering out the false predictions of JIT defect predictors. CP can filter out up to 100% of false negative predictions and 90% of false positives generated by CC2Vec, and up to 86% of false negative predictions and 83% of false positives generated by DeepJIT.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Formal methods**; *Software maintenance tools*.

KEYWORDS

Defect prediction, quality assurance, conformal prediction, machine learning, deep learning, correctness guarantees, uncertainty.

ACM Reference Format:

Xhulja Shahini, Andreas Metzger, and Klaus Pohl. 2024. An Empirical Study on Just-in-time Conformal Defect Prediction. In *21st International Conference on Mining Software Repositories (MSR '24)*, April 15–16, 2024, Lisbon, Portugal. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3643991.3644928>

1 INTRODUCTION

Just-in-time (JIT) defect prediction, a.k.a. change-level or commit-level defect prediction, aims to determine at check-in time whether a software change is defective or not [37, 44]. A defective software change is one that introduces one or more defects (a.k.a. faults or bugs) into the software [54]. JIT defect prediction helps to decide whether to perform targeted inspections and code reviews, as well as where and how to allocate testing efforts and resources [41, 58].

Compared with release-level defect prediction (e.g., see [46]), JIT defect prediction offers several benefits [21, 37, 55, 67]. (1) A JIT defect predictor provides predictions at a finer level of granularity, which enables practitioners to investigate fewer lines of code for defects. (2) A JIT defect predictor provides predictions immediately after committing the change to the code repository, thereby providing practitioners with real-time feedback.

Decision-making under uncertainty. Acquiring real-time feedback necessitates practitioners to make decisions on whether to trust the current prediction and take action. However, the accuracy of this decision is influenced by prediction uncertainty.

State-of-the-art JIT defect predictors have delivered substantial improvements in prediction accuracy; e.g., via better software metrics [3, 38], better representations of the data [15, 59], ensemble learning [20, 49, 60], or deep learning [14, 45]. Still, predictions can never be 100% accurate [16, 52]. There remains some degree of uncertainty on whether a prediction is correct, i.e., a true prediction.

If practitioners trust a false prediction and act upon it, the action might have negative consequences. Trusting a false *positive* prediction (i.e., a clean code change wrongly classified as a defective one) means that practitioners unnecessarily spend time and effort on investigating commits that do not actually contain defects [40, 46]. This may ultimately lead practitioners to ignore the predictions of the defect prediction tools that produce many false positives [24]. Trusting a false *negative* prediction (i.e., a defective code change wrongly classified as a clean one) means that practitioners may overlook actual defects, requiring costly fixes in later stages of software development [18, 28, 40, 69].

Heuristic Uncertainty Quantification. To support decision-making under uncertainty, practitioners may be provided with additional information that quantifies the degree of uncertainty of an individual prediction. Additional information explored in defect prediction research includes providing the *posterior probability* in addition to the class label [14, 21, 45], extracting the *confidence score* of the rules of a decision tree [10, 17], and computing the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MSR 2024, April 2024, Lisbon, Portugal

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0587-8/24/04...\$15.00

<https://doi.org/10.1145/3643991.3644928>

prediction probability by analyzing the agreement of base models in an ensemble [60, 66].

However, these types of uncertainty quantification only provide a heuristic quantification of uncertainty. They do not provide guarantees about the model's confidence in the correctness of a prediction [23] and thus may not be helpful for decision-making. Indeed, they may rather deceive a practitioner into trusting a false defect prediction, thus making the wrong decision.

Rigorous Uncertainty Quantification. An intriguing approach to address the problem of quantifying prediction uncertainty is conformal prediction (CP) [43, 50]. CP can be combined with any prediction model that provides some heuristic notion of uncertainty, such as the ones mentioned above. CP uses a small amount of additional calibration data to convert the heuristic notion of uncertainty into a rigorous one [1, 42]. Instead of generating an output in the form of a single label, CP generates prediction sets that are guaranteed, with probability $1 - \alpha$, to contain the true label. In the optimal case, the prediction sets would consist of only one label.

Other potential solutions to rigorously quantify prediction uncertainty would have been the Probably Approximately Correct (PAC) theory and the Bayesian framework. PAC only works well with clean data [43], which is a concern in defect prediction due to the quality of defect datasets [7, 8, 26, 27, 29] and noise due to the data-gathering process [8, 13, 25, 53]. Bayesian methods require knowing the probability distributions of the variables in the data and may lead to invalid confidence guarantees otherwise [34, 43].

Contribution. We perform an empirical study to assess how far CP is applicable and useful for JIT defect prediction. We experimentally evaluate two state-of-the-art JIT defect prediction techniques, namely DeepJIT (which uses deep learning) [14] and CC2Vec (which uses representation learning) [15]. We experiment with two popular defect data sets OPENSTACK and QT, which were also used in the original evaluation of the techniques.

We address the following research questions:

RQ1: *For what percentage of predictions can CP provide informative results?* CP generates a prediction set for each input instance, i.e., each prediction made by a JIT defect predictor. While CP always gives guarantees that the prediction sets contain the true label with probability $1 - \alpha$, a prediction set of size 2 (containing both labels "defective" and "clean") is obviously not informative, since one cannot make a decision based on it. This RQ analyzes how often CP generates prediction sets of size 1 and thus can provide useful information for decision-making.

RQ2: *What is the distribution of the class labels for which CP provides informative results?* This RQ refines the results of RQ1 by analyzing the distribution of the class labels for prediction sets of size 1. Given the typically highly imbalanced defect datasets (e.g., the QT dataset contains 8% defective commits and the OPENSTACK dataset 13% defective commits), it might be that CP generates informative results only for the "clean" class label. Such results would limit the practical usefulness of CP, as correctly identifying defective commits and correctly identifying clean commits are of different importance and related costs in practice.

RQ3: *What percentage of false predictions can CP filter out?* RQ1 and RQ2 focus on the informative results (i.e., prediction sets of size 1) of CP. RQ3 focuses on the non-informative results (i.e., prediction sets of size 2). We analyze how often false defect predictions end up

in prediction sets of size 2. Prediction sets of size 2 are considered non-informative predictions and, thus not used for decision-making. Via this metric, we assess the usefulness of CP to filter out false predictions. As indicated above, practitioners might ignore defect predictions if a defect predictor produces many false predictions. We thus assess how many of the false predictions produced by the JIT defect predictors may be avoided when using CP.

Paper Outline Section 2 provides relevant background on conformal prediction. Section 3 describes the study design. Section 4 provides the study results and the validity risks. Section 5 presents a critical discussion. Section 6 discusses related work.

2 CONFORMAL PREDICTION

Conformal prediction (CP) is a model-agnostic technique that can be combined with any point-prediction model, in order to complement predictions with their respective confidence measures [1, 42]. A point-prediction model (for simplicity we will call it a predictor) can be any prediction model, for e.g. a statistical, machine-learning (ML), or a deep-learning (DL) model, that generates as output a label (in the case of classification tasks), or a number (in the case of regression tasks). CP answers the question "Based on the previously seen instances, how confident is a predictor that a prediction is correct, i.e., the prediction matches the ground truth"? CP takes as input a heuristic quantification of prediction uncertainty (Section 1) to compute rigorous uncertainty outcomes, i.e., confidence scores.

Using a calibration data set (a small set of historical data), a trained predictor, and a given error threshold α , CP learns how to generate prediction sets, that are *guaranteed* to contain the ground truth with probability $1 - \alpha$. The error threshold $\alpha \in [0, 1]$, represents the maximum acceptable error for conformal prediction (CP) to generate prediction sets that do not include the true label. This is a user-defined value, usually a small one [1]. In the optimal case, the prediction sets would consist of a single label (in a classification task) or a very narrow interval (in a regression task).

The original CP algorithm is Transductive CP (TCP). However, TCP is computationally expensive [43]. In our study, we thus use an approximation version of it, namely inductive conformal prediction (ICP). We explain both these algorithms below.

Transductive CP. CP works for both classification and regression tasks. In this study we apply CP to a classification task, thus, for simplicity, we will explain CP in the context of a classification task.

Provided a dataset composed of n training instances and m test instances, whose possible data classes C are known beforehand, CP uses the training set to train a predictor $\hat{f}(x) \in [0, 1]^{|C|}$, which, for a given instance x , generates the probability distribution of x over the possible classes C .

To compute the calibration step, a non-conformity function $g(x, \hat{y})$ should be defined, which is used to quantify the level of disagreement between the knowledge of the predictor $\hat{f}$ and the true label y , given x [43]. It basically measures *confidence level* of how much the predicted $\hat{y}$ conform to the actual y .

TCP algorithm is as follows:

Step 1: Using $\hat{f}$, for every instance (x_i, y_i) in the training set, CP generates a set of scores, representing the probabilities of x_i belonging to any of the C possible classes.

Step 2: CP extracts the probability score $\hat{y} = \hat{f}(x_i)y_i$ corresponding to the true label y_i and computes the non-conformity/confidence score via $g(x, \hat{y})$.

Step 3: The confidence scores of all the instances in the training set are then used to compute the empirical confidence quantile. CP first calculates $\hat{q}$ via the following function:

$$\hat{q} = \frac{\lceil (n+1)(1-\alpha) \rceil}{n} \quad (1)$$

, where n is the number of instances in the training set and $\lceil x \rceil$ is the ceiling function of x . Then, CP extracts the $\hat{q}$ -th quantile of the confidence scores to compute the empirical confidence quantile.

Step 4: The empirical quantile is then used to construct prediction sets S , given newly unseen data instances. Typically, to assess the validity of the CP algorithm, one uses a dedicated test set, whose instances have not been used for training the predictive model. For each test instance $x_t \in X_{test}$ CP constructs a prediction set:

$S(x_t) = \{\hat{y}_c : \hat{f}(x_t)\hat{y}_c \geq 1 - \hat{q}\}$, by including only those labels $\hat{y}_c \in C$ whose probability score is larger or equal to $1 - \hat{q}$ [1].

The generated prediction set is guaranteed to provide the desired coverage, satisfying [57]:

$$\mathbb{P}(Y_{test} \in S(X_{test})) \geq (1 - \alpha) \quad (2)$$

, meaning that the prediction set S , is guaranteed, with a probability of at least $1-\alpha$, to contain the true prediction y_{test} of an input instance x_{test} .

In the case of a binary prediction task:

- if the prediction model is confident about the correctness of the prediction, only one class label will be included in the prediction set (set size=1).
- if the prediction model is not confident about the correctness of the prediction, both labels will be included in the prediction set (set size=2).

Inductive CP. The transductive CP algorithm computes all the above-mentioned steps of the CP algorithm for every instance in the test set. As such, it is highly computationally extensive. To enable the usage of CP algorithms for ML or DL models, which take a lot of computational resources and time, Papadopoulos et al. proposed ICP [42].

ICP stands for inductive conformal prediction. ICP first splits the training set into 2 subsets, namely training and calibration subsets. The training set is used to train the predictor $\hat{f}(x)$, which generates prediction scores for all the classes, given input instance x . The calibration set to perform the calibration, i.e., to train the conformal predictor on constructing prediction sets, ICP trains the predictor $\hat{f}$ using the training set and computes the empirical confidence quantile $\hat{q}$ using the calibration set.

In Equation 1, n represents the number of instances in the calibration set, as opposed to the number of instances in the training set for TCP. ICP performs the calibration step only once, and the same generated empirical quantile can be used to construct the prediction sets for each new instance.

ICP shows a great improvement in terms of runtime efficiency, in exchange for a slight decrease in terms of coverage [42, 43]. Thus, the coverage is expected to be *almost exactly* $1-\alpha$, instead of *exactly* $1-\alpha$. This property is known as marginal coverage [1].

CP correctness. Vovk states that conformal prediction must meet two desiderata: it should be *efficient* and *valid* [56]. This means that the prediction sets should be as small as possible for them to be *efficient*, and they should provide coverage of at least $1-\alpha$, for them to be *valid*.

To check the correctness of the CP algorithm one can use adaptivity to quantify its efficiency, and coverage to quantify validity. *Adaptivity*: describes the possible values and frequencies of the prediction set sizes. Prediction set sizes should be adaptive, meaning that the prediction set size should be as small as possible for "easy" inputs, and larger for "difficult" inputs. In the case of binary prediction tasks, the prediction sets can be of either size=1 or size=2.

Coverage: is an important measure of validity for CP. The goal of CP is for every given input, to construct a prediction set that is guaranteed, with a probability of almost exactly $1-\alpha$ to contain the ground truth label. Thus, to check the validity of the ICP algorithm, one needs to show that it provides the expected *marginal* coverage. Coverage can be measured for all the instances on the dataset, or for different classes separately.

Marginal coverage assesses coverage across all data instances, irrespective of their class. The term "marginal" is used to account for the stochastic impact of selecting instances for the calibration and test sets. The coverage results are computed and averaged over different runs, i.e., the marginal probability is estimated. Slight fluctuations are normal due to the lack of infinite data and experiments, as well as the stochastic nature of generating the calibration set [1].

Unlike marginal coverage, *class-conditional coverage* aims at ensuring almost $1-\alpha$ coverage in each of the classes in the dataset [1]. In the case of binary classification, given an error threshold of $\alpha = 0.05$, the class-conditional coverage aims at providing a coverage of 95% for the "clean" (class=0) data samples and 95% for the "defective" class=1 data samples. Nevertheless, unlike marginal coverage, class-conditional coverage is a much stronger assumption that cannot always be achieved. To compute class-conditional coverage, CP first splits the calibration set instances into classes and performs the calibration step for each class separately. If the data set consists of two classes, two different non-conformity quantiles will be computed, one for each class. For the instances on the test set, whose true class is not known, CP will compute the prediction sets using both non-conformity quantiles.

When computing class-conditional coverage for binary classification tasks, the prediction sets will result in sets of size 1 (including the respective class label) or 0 (no labels).

3 STUDY DESIGN

Below we introduce the design of our empirical study, including the evaluated JIT defect prediction techniques and used data sets.

3.1 JIT Defect Prediction Techniques

We experiment with two state-of-the-art JIT defect prediction techniques, namely DeepJIT and CC2Vec. We chose these techniques because they are published in top-tier conferences and have a considerable number of citations (CC2Vec-86, DeepJIT-104). We refer to the citation number in Scopus ¹. Both DeepJIT and CC2vec have

¹<https://www.scopus.com>

been featured as baselines in several JIT defect prediction papers [36, 39, 45, 64, 65].

3.1.1 DeepJIT. DeepJIT utilizes code, process, and developer metrics as input data for training a Convolutional Neural Network (CNN) [14]. Examples of the used metrics are size and diffusion metrics extracted from source code, commit messages, number of added/deleted lines of code extracted from change history, and developer metrics such as author/reviewer experience metrics. DeepJIT makes use of natural language processing to pre-process the natural-language input, i.e., the commit messages and code alterations, and then feed them into the CNN model for training. The trained model is used to make predictions on whether a commit is likely to be defective or clean. To address the imbalance in input data, DeepJIT employs a cost-sensitive learning approach. In practical terms, this means that the prediction model is penalized more heavily when it erroneously categorizes a defective commit as a clean commit, thereby forcing the prediction model to perform well also for the instances in the minority class.

DeepJIT's performance is evaluated with QT and OPENSTACK, resulting in an AUC = 0.765 for QT, and an AUC = 0.771 for OPENSTACK².

3.1.2 CC2Vec. Hoang et al. propose CC2Vec, a feature representation technique for software code changes, which can be used for different software development tasks [15]. In particular, they combine this feature representation with DeepJIT to enhance the predictive performance of DeepJIT. For CC2Vec the authors leverage artificial neural networks, specifically a hierarchical attention network, to generate an efficient representation of code change artifacts. CC2Vec uses two main code change artifacts, the log messages and the added/removed lines of code. The first sentence of a log message is used to extract the semantics of the code change. The added/removed lines of code are used to generate a vector representation of the change. CC2Vec employs an attention mechanism to capture the structural dependency of the code change, thus generating a better representation of the change.

Hoang et al. evaluated CC2Vec on QT and OPENSTACK: They report an AUC = 0.822 for QT, and an AUC = 0.809 for OPENSTACK³.

3.2 Datasets

As datasets, we use QT and OpenStack (collected by McIntosh and Kamei [33]) because they are widely used in DP research and were used to evaluate both DeepJIT and CC2Vec. Both datasets consist of different types of code, process, and author metrics, such as code size and diffusion metrics, change history metrics, author/reviewer experience metrics as well as review process metrics.

The QT dataset⁴ consists of data gathered from cross-platform development of desktop and embedded systems software. The QT dataset contains a total of 25,150 commits, 8% of them labeled as "defective".

The OPENSTACK dataset⁵ consists of data from the cloud computing infrastructure, which is open-source. The OPENSTACK dataset contains 12,374 commits 13% of them labeled as "defective".

²The dataset & the trained models are available at: <https://zenodo.org/records/3965246>

³The implementation of CC2Vec: <https://github.com/CC2Vec/CC2Vec>

⁴<https://www.qt.io/>

⁵<https://www.openstack.org/>

3.3 Algorithm and Parameterization

As introduced in Section 2, we employ inductive conformal prediction and in particular use the algorithm presented in [43].

The ICP algorithm is relatively generic and needs to be parameterized before it can be applied to a specific problem.

Calibration set size n . This parameter represents the number of data instances used for calibrating the ICP. Even though CP guarantees hold for any calibration set size, the random selection of the instances for constructing the calibration set might cause variability among the results of different runs (repetitions of the same experiment). Thus a large enough conformal set size could be necessary for accounting for variability [1]. Angelopoulos & Bates argue that a set size of 1000 instances is sufficient for ensuring coverage [1]. In this work [56], Vovk uses a calibration set of 999 instances. Thus, we set the $n = 1000$. Our experimental datasets consist of training and test sets. We keep the original set size for our test sets (1331 instances). We randomly selected n instances to create the calibration set. Note that the training, calibration, and test sets are disjoint sets.

User-specified error threshold α . The error threshold represents the maximum error that the CP can exhibit when computing confidence scores and prediction sets. Meaning, CP should guarantee, with a probability of $1 - \alpha$, that the prediction set contains the true label. The desired error threshold typically depends on the application domain. For example, high-risk or high-cost domains require higher prediction confidence and, thus lower error threshold. Other domains can be a bit more flexible regarding the error threshold. To assess the performance of CP under different settings, we treat α as an independent variable in our experiments, therefore, we experiment with three error threshold values, namely $\alpha \in \{0.05, 0.1, 0.15\}$. Thus, we provide the respective prediction coverages ($1 - \alpha \in \{95\%, 90\%, 85\%\}$). We argue that a coverage lower than 85% is not sufficient, thus we do not experiment with lower coverages.

Non-conformity score function $g(x, \hat{y})$. ICP requires a non-conformity score function, which quantifies the model's confidence in the correctness of a specific prediction. We use the same function as in [1, 56]. This is the typical score function used in the literature unless domain expertise indicates the usage of other score functions that better fit the data. Specifically, we define the non-conformity score to be $g(x, \hat{y}) = 1 - \hat{y}_{true}$, where x is the input instance and $\hat{y}_{true}$ is the defect predictor's sigmoid score corresponding to the true label of x .

3.4 Independent variables

Our experiments consider three independent variables:

- The JIT defect prediction technique: DeepJIT & CC2Vec
- The dataset: QT & OPENSTACK
- The error threshold: $\alpha \in \{0.05, 0.1, 0.15\}$

We perform experiments for all the possible combinations of the independent variables, resulting in a total of 12 experimental settings. Each experimental setting is repeated 100 times as suggested by [1, 42], resulting in a total of 1200 experiments.

3.5 Dependent variables

As dependent variables, we measure the following metrics for the different research questions.

RQ1 - Rate of informative CP predictions: we compute the rate of the instances for which the CP algorithm can provide informative correctness guarantees, in the whole test set. To do so we compute:

$$Inf = \frac{m_{s=1}}{n}$$

, where $m_{s=1}$ is the number of instances for which CP generates a prediction set of size 1, and n is the total number of instances in the test set.

RQ2 - Distribution of informative CP predictions: For all the informative CP predictions obtained in RQ1, we investigate the percentages of instances belonging to the class "defective" (class=1) and "clean" (class=0) respectively.

$$Inf_{c=0} = \frac{m_{s=1,c=0}}{m_{s=1}}, \quad Inf_{c=1} = \frac{m_{s=1,c=1}}{m_{s=1}}$$

, where $m_{s=1,c=0}$ is the number of instances for which CP generates a prediction set of size 1 consisting of a class label=0, $m_{s=1,c=1}$ is the number of instances for which CP generates a prediction set of size 1 consisting of a class label=1, and $m_{s=1}$ is the total number of instances for which CP generates a prediction set of size 1.

RQ3 - The rate of filtered-out FP and FN: we measure the rate of false positive (FP) & false negative (FN) predictions that CF filters out. We first compute FP_{tot} and FN_{tot} . That is the total number of FP and FN predictions generated by the selected JIT defect predictors. Next, we compute $FP_{filtered}$ and $FN_{filtered}$. Basically, we count for which percentage of the FP & FN predictions generated by the JIT-DP technique, CP generates prediction sets of size 2, i.e., the prediction's confidence is lower than the desired confidence level. To obtain the filtered FP&FN rate we compute:

$$filteredFP = \frac{FP_{filtered}}{FP_{tot}}, \quad filteredFN = \frac{FN_{filtered}}{FN_{tot}}$$

4 RESULTS

Here we report the results of our research questions. We provide results averaged over the 100 repetitions for each experimental setting and also provide their standard deviation. Results of the individual experiments are available in our repository, as provided in Section 8. We first describe the results with DeepJIT, then the results with CC2Vec, followed by the summary of the RQ. We follow the same structure for all our research questions.

4.1 RQ1: Rate of informative CP predictions

Table 1: DeepJIT: Rate of Informative CP Predictions

α	OPENSTACK			QT		
	% _{ss=1}	Min-max	STDev	% _{ss=1}	Min-max	STDev
(0.05)	27%	175-550	87	28%	187-563	83
(0.10)	46%	398-682	41	52%	480-910	92
(0.15)	55%	601-807	29	70%	748-1134	84

4.1.1 DeepJIT. Table 1 shows the rate of informative predictions (%_{ss=1}, prediction set size=1), that CP generated for DeepJIT, with different error thresholds and different datasets. Min-max shows the minimum vs. the maximum number of instances resulting in a

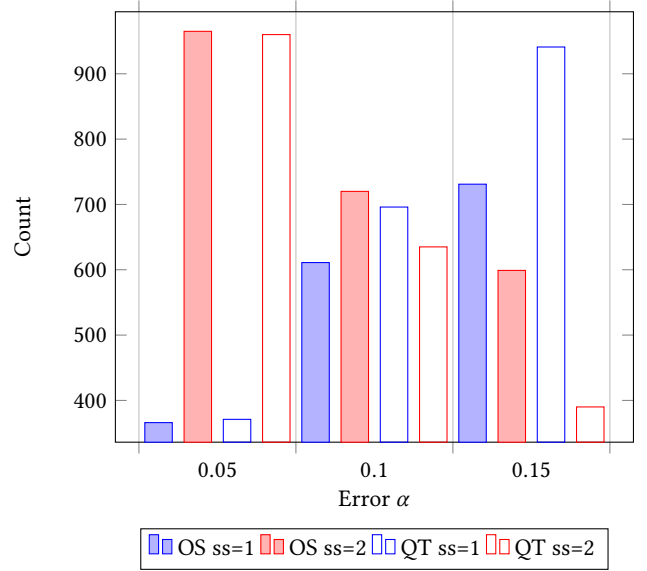


Figure 1: Prediction Set Sizes (ss) for DeepJIT

prediction set of size one, among the 100 runs. STDev shows the standard deviations of the 100 runs.

As we can observe, when augmenting DeepJIT predictions, CP generates informative prediction sets 27% resp. 28% of the time for an α of 0.05. With a much higher margin for error ($\alpha = 0.15$), CP generates informative prediction sets 55% resp. 70% of the time.

DeepJIT reported an AUC score of 0.771, when trained with the OPENSTACK dataset, and 0.765, when trained with the QT dataset. But if we are only interested in predictions whose correctness we can guarantee, we observe that only for a small fraction of DeepJIT predictions CP estimates confidence scores that surpass the desired confidence level, i.e., informative prediction sets.

The histogram in Figure 1, gives the absolute number of informative (set size=1) and non-informative (set size=2) predictions for different error thresholds α , for both datasets (OS stands for OPENSTACK). We observe that for large values of α , CP delivers a relatively higher rate of informative predictions for the QT dataset than for the OPENSTACK dataset.

To analyze why, we examine the mean, maximum, and standard deviation (Table 1). Comparing QT and OPENSTACK, we notice that the maximum is considerably larger (1134 vs. 807), as is the standard deviation (84 vs. 29). The test set size is constant for all experiments. The higher spread of values indicates that results for the QT dataset vary much more between experiment runs. This higher variance might be attributed to the larger data set size and more pronounced data imbalance of the QT dataset in comparison to the OPENSTACK dataset, affecting the predictor's decision in a prediction. This could lead to the occurrence of the defect predictor performing the so-called label flip, in different training runs. In simpler words, the model changes the prediction for the instances for which the model is unsure, i.e., instances that lie close to the decision boundary, but it does not affect the instances that lay far from the decision boundary. Thus for smaller error rates, we observe little variance, which also shows that only a small fraction

of the instances lay far enough from the decision boundary, thus the defect predictor can predict their label with confidence.

The variance of DeepJIT has also been reported in previous studies. Shahini et al. report an accuracy variance of up to 10.3% with $STDev = 2.72$ in terms of per-class accuracy metric for the "defective" class, for DeepJIT trained several times under identical settings [51].

4.1.2 CC2Vec. Table 2 shows the rate of informative predictions ($\%_{ss=1}$), that CP generated for CC2Vec, with different error thresholds and different datasets.

Like for DeepJIT, a larger error threshold allows CP to generate informative prediction sets up to 24% of the time.

CC2Vec reported an AUC score of 0.822 when trained with the OPENSTACK dataset, and 0.809, when trained with the QT dataset. But if we are only interested in predictions whose correctness we can guarantee, we observe that for an $\alpha = 0.05$ only for 9% of the predictions (for OPENSTACK) and 8% of the predictions (for QT) CP provides correctness guarantees. These values are even smaller than the results of DeepJIT, even though CC2Vec has reported higher accuracy than DeepJIT.

To investigate this discrepancy, we look at other accuracy metrics rather than AUC score, for both DeepJIT and CC2Vec.

AUC score is a threshold-independent metric, which is not always appropriate for evaluation, particularly in scenarios where a) prediction probabilities require calibration, and b) the cost of false negatives versus false positives is not the same. Both these cases are desirable for the goal of this study. To gain an understanding of the CC2Vec model's performance concerning both minority and majority classes, we calculate the precision, recall, F1-score, specificity, and FAR (false alarm rate). An interesting observation was that CC2Vec exhibits a very high recall (up to 99%) which means that it produces fewer false negatives, but it performs worse than DeepJIT in terms of precision and FAR (meaning it produces more false positives and fewer true negatives)⁶. DeepJIT achieves a precision=0.19 and FAR=0.32, meanwhile, CC2Vec achieves a precision=0.096, and FAR=0.67. These outcomes align with the insights from Pornprasit et al., who similarly report CC2Vec's tendency to classify instances as "defective" [45].

CC2Vec's tendency to generate false positives, results in CP assigning lower confidence scores to individual predictions, thereby leading to fewer informative sets. Thus, CC2Vec's improvement over DeepJIT in terms of AUC score is not reflected in the rate of informative predictions that CP generates.

Result: DeepJIT achieves an AUC of 0.771, and CC2Vec achieves higher accuracy (AUC=0.822). When using CP to provide correctness guarantees we observe that only 27% (for DeepJIT) and 9% (for CC2Vec) of the predictions can be guaranteed, with a probability of 95%, to be correct.

Accuracy alone does not imply practical usability.

⁶The rest of the results can be found in our supplementary material.

Table 2: CC2Vec: Rate of Informative CP Predictions

α	OPENSTACK			QT		
	$\%_{ss=1}$	Min-max	STDev	$\%_{ss=1}$	Min-max	STDev
(0.05)	9%	92-170	14	8%	79-160	20
(0.10)	16%	158-299	23	17%	173-283	24
(0.15)	24%	262-433	26	24%	263-456	41

4.2 RQ2: Distribution of Informative CP Predictions

In this RQ we look deeper into the informative prediction sets that we obtained in RQ1, to investigate the distribution of informative CP results across the class labels "defective" and "clean" respectively.

Table 3: DeepJIT: Class Distribution of Informative CP Predictions

α	OPENSTACK			
	#defective	#clean	%defective	%clean
(0.05)	27	260	9	91
(0.10)	51	417	11	89
(0.15)	74	437	14	86

α	QT			
	#defective	#clean	%defective	%clean
(0.05)	27	273	9	91
(0.10)	45	505	9	91
(0.15)	57	657	8	92

4.2.1 DeepJIT. Table 3 shows the distribution of informative CP results (prediction sets of size 1) across both class labels. The left side of the table shows the exact numbers of the instances whose true class is "defect" and "clean". The right side of the table shows the respective percentages.

Our findings reveal that CP generates informative prediction sets that consist of approximately 12% of instances belonging to the class "defective" for OPENSTACK dataset and approximately 9% of input instances belonging to the class "defective" for QT.

While these results might seem highly imbalanced, it is important to consider the highly imbalanced nature of our experimental datasets. For context, the OPENSTACK dataset comprises 13% "defective" instances and 87% "clean" instances, while the QT dataset consists of 8% "defective" instances and 92% "clean" instances.

To gain deeper insights into our results, we examine class-conditional coverages (CCC) (see Section 2) to analyze how far can CP deliver coverage for the different labels.

Due to the lack of space, the rest of the figures and results can be found in our supplementary files on Github. The first thing that we notice is that for QT and OPENSTACK, CP achieves almost exactly the expected class-conditional coverage for both classes.

For instance, with the OPENSTACK dataset and $\alpha = 0.05$, CP achieves a CCC of 0.94 for the subset of instances labeled as "clean" and 0.97 for the subset of instances labeled as "defective." In the case of the QT dataset, CP achieves a CCC of 0.95 for the subset of "clean" instances and 0.94 for the subset of "defective" instances.

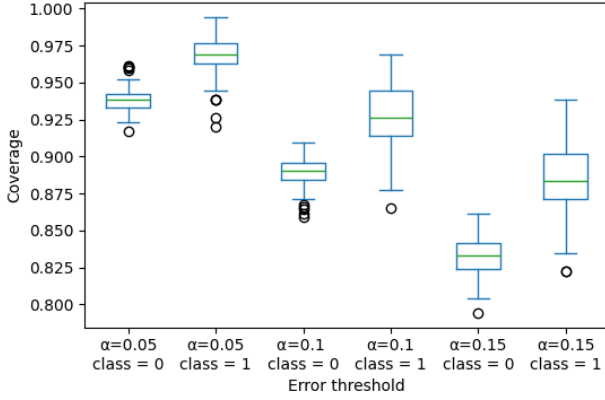


Figure 2: OPENSTACK: CP Class-conditional Coverage

Upon closer examination of class-conditional coverage, we observe two interesting phenomena:

1: *Dispersion in Coverage Values.* Analyzing individual experiments, we observe that for both datasets, CP delivers coverage values exhibit a higher dispersion (STDev) for instances labeled "defective" (e.g.: for QT with $\alpha = 0.05$, $STD = 0.04$) compared to instances labeled "clean" ($STD = 0.007$). This observation supports our hypothesis that dataset imbalance may contribute to increased variability in the model's accuracy and calibration set samples.

2: *Higher Coverage for "Defective" instances in OPENSTACK.* For the OPENSTACK dataset, we observe higher coverage for the "defective" class instances, compared to the QT dataset. A possible explanation could be the fact that OPENSTACK has a larger percentage of "defective" class instances than the QT dataset. As such the calibration subsets have a higher chance of containing enough instances of the minority class, leading to a more accurate estimation of the confidence quantile, and thus better coverage of this class.

4.2.2 **CC2Vec.** Table 4 shows the distribution of informative CP results across the class labels, for CC2Vec.

Table 4: CC2Vec: Class Distribution of Informative CP Predictions

α	OPENSTACK			
	#defective	#clean	%defective	%clean
(0.05)	36	1	97	3
(0.10)	64	2	97	3
(0.15)	82	6	93	7

α	QT			
	#defective	#clean	%defective	%clean
(0.05)	32	1	97	3
(0.10)	56	9	86	14
(0.15)	70	26	73	27

Our results show that CP generates informative prediction sets that consist of approximately 94% of instances belonging to the class "defective", for OPENSTACK dataset; and $\sim 80\%$ of the instances belonging to the class "defective", for the QT dataset.

When compared to the experiments with DeepJIT, we notice that the results are almost flipped. As discussed in RQ1, DeepJIT and CC2Vec behave very differently in terms of the different errors. DeepJIT delivers higher precision, meanwhile, CC2Vec delivers much higher recall.

CC2Vec's tendency to classify instances as "defective" appears to influence the class distribution of informative CP results, resulting in a higher percentage of informative prediction sets for the class "defective". Whenever CC2Vec generates a false positive, CP assigns a low confidence score to that prediction, resulting in a prediction set of size 2. Consequently, when the predictive model exhibits a tendency to classify instances into a specific class, CP seeks to mitigate this tendency by reducing the overall number of informative sets. As observed in RQ1, CP generates fewer informative predictions for CC2Vec. We also make the assumption that CC2Vec's inclination to classify instances as "defective" causes CP to commit more errors on false positive instances.

For this reason, we further examine CP errors. More specifically, we analyze instances where CP generated a prediction set of size 1 containing the incorrect label. Remember that CP can make errors with α probability. As anticipated, for CC2Vec, CP predominantly makes errors on false positive predictions. For CC2Vec trained with QT, CP commits 100% of errors on false positive instances across all values of α . Similarly, for CC2Vec trained with OPENSTACK, CP commits errors on 95-100% of false positive instances for all α values.

Result: CP generates informative predictions that cover both the minority and the majority class labels. Nevertheless, dataset imbalance and the tendency of a JIT predictor to classify instances into a particular class will influence the class distribution of the informative sets, reflecting and aligning with that specific tendency.

4.3 RQ3: Filtered-out False Predictions

Table 5: False predictions of DeepJIT & CC2Vec

	DeepJIT				CC2Vec			
	QT		OPENSTACK		QT		OPENSTACK	
	FP	FN	FP	FN	FP	FN	FP	FN
(Avg)	354	34	529	25	808	3	812	7
(Min)	304	29	437	12	601	0	625	2
(Max)	414	38	658	42	1006	8	981	12
(STDev)	21.5	1.8	46.2	6.3	83.9	1.6	68.9	2.1

Different from the previous RQs, in this research question we investigate the instances for which CP generates a prediction set of size 2, thus leading to practitioners rejecting the prediction of the JIT predictor. Below we present the average rate of false positives and false negatives (generated by DeepJIT and CC2Vec) that CP filters out. We show the results for the different error thresholds we experimented with, and the different datasets used for evaluating the model.

4.3.1 **DeepJIT.** Table 5 outlines the rate of false positive (FP) and false negative (FN) predictions that DeepJIT produces when trained with QT and OPENSTACK. When DeepJIT is trained with the QT

Table 6: DeepJIT: Filtered-out False Predictions

α	OPENSTACK				QT			
	% FP	STDev	%FN	STDev	%FP	STDev	%FN	STDev
(0.05)	86	0.02	82	0.06	81	0.03	83	0.07
(0.10)	74	0.03	65	0.07	63	0.05	57	0.1
(0.15)	60	0.04	62	0.08	42	0.06	34	0.09

dataset, it produces an average of 354 FP and 34 FN predictions. Conversely, when trained with the OPENSTACK dataset, it yields an average of 529 FP and 25 FN. predictions. As can be seen from Table 5, DeepJIT exhibits a higher variance, in terms of FP & FN, when trained with OPENSTACK (STDev=46.2) compared to QT (STDev =21.5). This observation also supports our above-mentioned insights regarding the effect of data imbalance on the performance of the JIT defect predictor.

Table 6 outlines the rates of FP and FN generated by DeepJIT, which can be filtered out by CP. Notably, CP demonstrates its effectiveness in reducing both false positive and false negative predictions, especially for small values of α . As anticipated, with an increasing error threshold, CP becomes less effective in filtering out these errors.

An interesting observation is the similarity in the percentages of filtered-out FP & FN predictions, indicating that CP could be valuable in practice for mitigating both types of errors. It is crucial to emphasize that CP, with a probability of $1-\alpha$, guarantees that the true label is within the predicted set. Adhering to the error tolerance α , CP does not filter out 100% of the FN & FP. If we set a smaller α , CP would filter more false predictions, but at the cost of producing fewer informative prediction sets.

4.3.2 CC2Vec. Table 5 outlines the rate of false positive (FP) and false negative (FN) predictions that CC2Vec produces when trained with QT and OPENSTACK. When trained with the QT dataset, CC2Vec produces an average of 808 FP and 3 FN predictions. Conversely, when trained with the OPENSTACK dataset, it yields an average of 812 FP and 7 FN. predictions. Compared to DeepJIT, CC2Vec produces less FN, but also more FP. This further supports our observation in RQ1 & RQ2 about the tendency of CC2Vec to produce false positives, significantly more than DeepJIT.

Table 7: CC2Vec: Filtered-out False Predictions

α	OPENSTACK				QT			
	% FP	STDev	%FN	STDev	%FP	STDev	%FN	STDev
(0.05)	90	0.02	100	0	90	0.02	95	0.2
(0.10)	81	0.03	99	0.02	80	0.03	95	0.2
(0.15)	72	0.03	99	0.02	71	0.04	95	0.2

Table 7 shows the ratio of FP and FN predictions of CC2Vec, that CP can filter out. We notice that CP can filter out up to 100% (for OPENSTACK) and 90% (for QT) of the FN predictions, when $\alpha = 0.05$, and 95-99% of the FN for larger values of α . This outcome is expected, considering that CC2Vec yields very few false negatives.

In terms of false positive (FP) predictions, CP demonstrates the capability to filter out up to 90% of FP instances for both OPENSTACK and QT when $\alpha = 0.05$. Given the significantly high number

of false positives generated by CC2Vec, this result underscores CP’s effectiveness in filtering out false positives as well. Overall, these findings suggest that CP performs well in mitigating both false positive and false negative predictions generated by CC2Vec.

Result: Our experiments demonstrate CP’s capability in filtering out both types of errors (FP & FN) produced by JIT defect predictors, suggesting that CP is a valuable approach for mitigating false predictions.

4.4 Validity risk

Internal validity The selection of the JIT- defect prediction techniques, might introduce bias in our results. Thus, we experiment with two different JIT defect predictors of different nature. DeepJIT leverages deep learning to improve prediction accuracy, and CC2Vec leverages representation learning to better represent the input data. CC2Vec is then combined with DeepJIT to improve defect prediction accuracy. We run the code of the original papers, to make sure that they deliver the accuracy levels reported in the original papers. We followed the replication documentation for the GitHub repositories of the respective JIT defect prediction technique to run the code. We obtained the same results as reported in the original publications.

The stochastic nature of the JIT defect predictors might introduce variance in the obtained results, affecting the internal validity. To account for this risk, we repeat each experiment 100 times and we make available the results of all individual experiments.

Construct validity CP requires data exchangeability. Exchangeability is a weaker notion of independent identically distributed (i.i.d.). In simple terms, exchangeability means that the order of the variables does not affect their joint distribution [43]. Given that i.i.d. is a requirement for applying ML to a dataset, we assume that our data are also exchangeable since every i.i.d sequence is exchangeable [50]. Nevertheless, we follow the guidelines of Shafer & Vovk by randomly permuting all variables, to create an order in which the sequences become inherently interchangeable. According to the law of large numbers, as we traverse sequences in the permuted order (to calibrate CP and evaluate it), CP commits errors at a consistent rate, approximately equal to the expected α [50].

Another risk can arise from the usage ICP algorithm, which is an approximation version of TCP. As such it might not provide exactly $1-\alpha$ coverage. Vovk et al. show that ICP offers enhanced computational efficiency while maintaining (almost) identical coverage compared to the original CP algorithm [43]. They state that the choice of the algorithm does not affect the validity of the obtained results, for as long as we can show that it achieves the desired marginal coverage $1 - \alpha$, and it is adaptive (see Section 2). To account for this risk we validate our ICP implementation by checking for adaptivity and coverage [1]. We repeat the experiments 100 times and plot the results of the coverage for all our runs. We expect the histogram of the coverage values to be centered around approximately $1 - \alpha$. Slight variations in the observed coverage are normal, due to the fact that our data and the number of repetitions are finite [1]. Our results (see supplementary documents) show that the ICP algorithm delivers the expected adaptivity and marginal coverage. The histogram in Figure 1, shows that our conformal prediction

model produces prediction sets with different sizes, indicating that the implemented ICP is adaptive, and thus working as expected.

External validity We only experimented with two datasets and two JIT defect prediction techniques, thus we can not ensure generalizability with other JIT defect prediction techniques and datasets.

5 DISCUSSION

Our results suggest that obtaining guarantees for JIT defect prediction techniques, with very small residual error, comes at the cost of rejecting many predictions for which prediction certainty can not be established.

On one hand, these outcomes suggest that prediction accuracy might not be sufficient to properly evaluate JIT defect predictors. On the other hand, the predictions that are retained, for which guarantees can be provided, may still be helpful for practitioners.

JIT defect prediction techniques. The outcomes of our experiments reveal that, for different JIT defect prediction techniques, CP generates distinct rates of informative predictions, even when applied to the same datasets. It is noteworthy that a higher AUC score does not necessarily correlate with more informative results. Additionally, we observe that the tendency of a JIT predictor to classify instances into a specific class significantly impacts the class distribution of informative CP results. This observation underscores the importance of considering the inherent tendencies of JIT predictors when evaluating the rate of informative prediction generated via CP.

Datasets. Our results suggest that for small α values, the differences in the dependent variables are negligible for different datasets. These results indicate that only for a small fraction of the instances, JIT defect predictors can make predictions for which CP assigns a high confidence score. Additionally, we observed that the dataset imbalance can be projected in the distribution of informative CP results across the class labels.

Error thresholds. With a larger α , we observe that CP generates more informative prediction sets, but also filters out fewer false predictions.

Table 8: Filtered-out True Predictions

α	DeepJIT with QT		CC2Vec with QT	
	mean (%)	STDev	mean (%)	STDev
(0.05)	48	62	37	79
(0.10)	39	65	34	71
(0.15)	17	56	32	56

CP is capable of filtering out false positives and false negatives, indicating that it might be a valuable tool for generating prediction guarantees in JIT defect predictors. However, it is important to recognize that CP does not filter out only false predictions but also discards some true predictions. Table 8 illustrates the mean rate and standard deviation of the true predictions of DeepJIT (on the left) and CC2Vec (on the right) that CP filters out for different values of α . Consequently, a thoughtful selection of the error threshold α is recommended, ideally tailoring it to the specific domain of the application. Furthermore, it might be beneficial to differentiate between various types of defects and establish an error threshold accordingly.

Another important consideration for the successful implementation of CP in JIT defect predictors is the choice of the non-conformity score function. In this study, we used a generic and commonly used non-conformity score function as a naive baseline. Customizing this function based on a better understanding of the data may yield improved results in terms of informative prediction sets. Thus leading to a more efficient filtering of the defect predictor true predictions. One might also differentiate between the type of errors, to obtain a more selective filtering of false and true predictions.

The JIT defect prediction models used in this study produce a probability (sigmoid) score that is used to make a prediction. We use this score as an input for our non-conformity score function. However, a better uncertainty quantification heuristic may better reflect the model's understanding of the data patterns, leading to a higher rate of informative prediction sets.

5.1 Future directions

Existing JIT defect prediction techniques continuously report competitive accuracies in research. Nevertheless, these techniques need to be evaluated in other dimensions apart from accuracy, to ensure take-off and acceptance in practice. No matter how good a new JIT defect predictor is, it will always exhibit some degree of uncertainty, leading to false predictions. False predictions may lead to enormous consequences in practice, thus researchers should consider how to evaluate JIT-defect prediction tools in more practical ways.

Recent approaches have proposed to also take into consideration the amount of effort needed to inspect the predicted defective change. These approaches are called effort-aware JIT defect prediction. They typically quantify effort in terms of the total number of changed lines of code and use this score to order or rank the software changes so as to prioritize defect density, or inspection effort, or to find a trade-off between them [21, 62, 67]. While considering effort in fault prediction is a useful aspect, the proposed approaches do not provide guarantees that the effort-aware JIT predictions are actually correct.

Complementing the individual predictions made by JIT defect predictors with confidence guarantees is desirable, to build trust for the practitioners.

The next step could then be to use explainable AI (XAI) to help developers better understand the predictions of these techniques. For e.g., using contrastive explanations might help practitioners understand the prediction and also guide them on how to fix the defect(s). In a study by [19], the authors found that practitioners regard contrastive explanations as convenient and helpful.

6 RELATED WORK

Researchers have proposed several approaches that aim to improve the accuracy of JIT defect predictors: *a)* by analyzing which input artifacts, i.e., proposing change metrics that are better predictors of defects [31, 32, 48], *b)* by introducing better data representation approaches [15, 68], *c)* by improving the quality of the datasets [7, 47], or *d)* by leveraging better ML/DL algorithms [14, 46, 61]. Nevertheless, these approaches improve accuracy but do not provide any correctness guarantees for the specific predictions.

Ensemble models have also gained great popularity in JIT-DP community. Yang et al. [60] and Zhang et al. [66] employ ensemble methods to improve within-project and cross-project JIT defect prediction accuracy. Aside from the improved accuracy and more stable results, ensemble methods can also provide prediction intervals instead of single-point prediction. However, intervals do not provide any guarantees about the accuracy of the predictions.

Mesquita et al. construct a defect predictor with a reject option [35]. They train two classification models and compare their predictions to derive the predicted label. If the models disagree they reject the prediction, i.e., no prediction is made. The authors extract the predictive model's output probability score and use it to represent the model's confidence. As mentioned above, this heuristic is not a reliable confidence score, and it can often be misleading.

Guo et al. address the uncertainty associated with automatically labeling commit changes by employing the widely recognized SZZ algorithm [11]. They introduce the CLI model, Confidence Learning Imbalance, which identifies labels predicted with low confidence and quantifies the level of uncertainty through a probability matrix. However, it's worth noting that the authors do not measure or address the uncertainty of the JIT defect predictor trained with the cleaned data.

Stepping away from just-in-time (JIT) defect prediction and examining the broader landscape of defect prediction research, various studies address uncertainty at distinct stages of constructing defect predictors. These stages include the data level, training level, and prediction level.

Research on uncertainty at the data level primarily addresses uncertainties introduced during the measurement and gathering process of the data. Azzeh et al. employ gray relational analysis (GRA) and fuzzy set theory (FST) to mitigate the level of uncertainty in defect datasets [2]. In other studies, a combination of uncertainty sampling and active learning is employed to manually label instances for which the predictive model exhibits high uncertainty about the correctness of the predicted label [30, 63].

Bowes et al. assess uncertainty at the training level by measuring the uncertainty of different classifiers in terms of label-flipping (i.e., changing the prediction), among 100 training runs [6]. Shahini et al. measure the uncertainty of defect predictors across identical training runs, stemming from the inherent non-deterministic introducing factors present in the ML models[51].

Research on uncertainty at the prediction levels primarily revolves around the selection of predictive models capable of computing a posterior probability as an estimate of uncertainty. Bayesian methods are commonly employed for this purpose [9, 12?]. Additionally, some studies focus on models that calculate prediction intervals to illustrate the uncertainty associated with a specific prediction [4, 5, 22].

7 CONCLUSION

In this study, we use conformal prediction (CP) to generate correctness guarantees for the predictions of two state-of-the-art JIT defect prediction techniques. Specifically, we make the following contributions:

- We demonstrate how conformal prediction can be applied to defect prediction, using 2 state-of-the-art defect prediction

techniques, to complement individual predictions with their respective confidence levels.

- We empirically evaluated how useful the selected defect prediction techniques are in practice by measuring the rate of the prediction whose correctness can be guaranteed, given a desired error threshold, α . We experiment with three different α values.
- We analyzed how far can CP provide coverage to both the majority and the minority classes of the data, to account for data imbalance.
- We measured the added value of complementing JIT defect prediction with CP by measuring how many false positives and false negative predictions can we avoid.
- We evaluate our results using two publicly available datasets.
- To support open science, we publish our implementation and results.

Our results show that achieving high accuracy does not necessarily mean that the predictions generated by these techniques are practically useful. Our experiments show that CP can ensure correctness with a 95% probability, for only 27% (for DeepJIT) and 9% (for CC2Vec) of the JIT defect predictions. While the rate of informative predictions that CP generates for the selected JIT defect predictors is not very high, these results are still better than having predictions without any guarantees. It can still be useful for a practitioner to obtain a fraction of predictions that are guaranteed with a probability of 95% to be correct.

CP generates informative predictions that cover both the minority and the majority class labels. Nevertheless, *a)* dataset imbalance and *b)* the tendency of a JIT predictor to classify instances into a particular class will influence the class distribution of the informative sets, reflecting and aligning with that specific tendency.

Lastly, our experiments indicate that CP might be a valuable technique for filtering out the false predictions of JIT defect predictors. CP can filter out up to 100% of false negative predictions and 90% of false positives generated by CC2Vec, and up to 86% of false negative predictions and 83% of false positives generated by DeepJIT, as such it may be valuable in practice.

Based on this study outcomes, we strongly recommend researchers and practitioners to consider combining conformal prediction with just-in-time (JIT) defect predictors for improved recall and precision. The integration of conformal prediction with JIT defect predictors does not impose significant additional efforts in the development of a JIT defect predictor. However, the provision of predictions accompanied by correctness guarantees can be instrumental in minimizing costs associated with defects.

7.1 Future work

In our future work, we aim to enhance the generalizability of this study by expanding our assessment to include other just-in-time (JIT) defect prediction techniques. Additionally, we plan to explore alternative methods to increase the rate of informative predictions generated by conformal prediction (CP) and particularly minimize the rate of filtered true predictions, by experimenting with different non-conformity functions.

Moreover, our future work will involve utilizing CP not only for assessment but also as a complementary approach to JIT defect

predictors. Thus, we plan to explore different ways how to leverage CP for its capabilities in providing confidence guarantees and improving the overall robustness of JIT defect predictors.

8 DATA AVAILABILITY

To adhere to the open science policy we make our implementation, additional scripts, and obtained results used in this study publicly available. For our experiments, we use two publicly available datasets⁷. Our implementation and experimental results are made available in GitHub⁸.

REFERENCES

- [1] Anastasios N. Angelopoulos and Stephen Bates. 2023. Conformal Prediction: A Gentle Introduction. *Foundations and Trends® in Machine Learning* 16, 4 (2023), 494–591. <https://doi.org/10.1561/2200000101>
- [2] Mohammad Azzeh, Yousef Alqasrawi, and Yousef Elsheikh. [n. d.]. A soft computing approach for software defect density prediction. *Journal of Software: Evolution and Process* [n. d.], e2553. <https://doi.org/10.1002/smr.2553>
- [3] Jacob Barnett, Charles Gathuru, Luke Soldano, and Shane McIntosh. 2016. The relationship between commit message detail and defect proneness in Java projects on GitHub. 496–499. <https://doi.org/10.1145/2901739.2903496>
- [4] Momotaz Begum and Tadashi Dohi. 2016. Prediction Interval of Cumulative Number of Software Faults Using Multilayer Perceptron. 619 (01 2016), 43–58. https://doi.org/10.1007/978-3-319-26396-0_4
- [5] Momotaz Begum, Mohammad Hossain, Md. Jakirul Islam, and Md. Samzid Hafiz. 2021. Long-term Software Fault Prediction with Robust Prediction Interval Analysis via Refined Artificial Neural Network (RANN) Approach. *Engineering Letters* 29 (08 2021), 44.
- [6] Tracy Hall David Bowes and Jean Petrić. 2017. Software defect prediction: do different classifiers find the same defects? *Software Quality Journal* 26, 2 (2017), 525–552. <https://doi.org/10.1007/s11219-016-9353-3>
- [7] Ruifeng Duan, Haitao Xu, Yuanrui Fan, and Meng Yan. 2021. The impact of duplicate changes on just-in-time defect prediction. *IEEE Transactions on Reliability* 71, 3 (2021), 1294–1308.
- [8] Yuanrui Fan, Xin Xia, Daniel Alencar da Costa, David Lo, Ahmed E. Hassan, and Shanping Li. 2021. The Impact of Mislabeled Changes by SZZ on Just-in-Time Defect Prediction. *IEEE Transactions on Software Engineering* 47, 8 (2021), 1559–1586. <https://doi.org/10.1109/TSE.2019.2929761>
- [9] N. Fenton, P. Krause, and M. Neil. 2002. Software measurement: uncertainty and causal modeling. *IEEE Software* 19, 4 (2002), 116–122. <https://doi.org/10.1109/MS.2002.1020298>
- [10] Yoav Freund and Llew Mason. 1999. The Alternating Decision Tree Learning Algorithm. In *ICML*. 124–133. <http://www.lismason.com/papers/ICML99-AlternatingTrees.pdf>
- [11] Shikai Guo, Dongmin Li, Lin Huang, Sijia Lv, Rong Chen, Hui Li, Xiaochen Li, and He Jiang. 2023. Estimating Uncertainty in Labeled Changes by SZZ Tools on Just-In-Time Defect Prediction. *ACM Trans. Softw. Eng. Methodol.* (dec 2023). <https://doi.org/10.1145/3637226>
- [12] María José Hernández-Molinos, Angel J. Sánchez-García, Rocío Erandi Barrientos-Martínez, Juan Carlos Pérez-Arriaga, and Jorge Octavio Ocharán-Hernández. 2023. Software Defect Prediction with Bayesian Approaches. *Mathematics* 11, 11 (2023). <https://doi.org/10.3390/math11112524>
- [13] Kim Herzig, Sascha Just, and Andreas Zeller. 2015. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering* 21 (04 2015). <https://doi.org/10.1007/s10664-015-9376-6>
- [14] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc (Eds.). IEEE / ACM, 34–45. <https://doi.org/10.1109/MSR.2019.00016>
- [15] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. Cc2vec: Distributed representations of code changes. *Proceedings - International Conference on Software Engineering* (2020), 518 – 529. <https://doi.org/10.1145/3377811.3380361> All Open Access, Green Open Access.
- [16] Eyke Hüllermeier and Willem Waegeman. 2021. Aleatoric and epistemic uncertainty in machine learning: an introduction to concepts and methods. *Mach. Learn.* 110, 3 (2021), 457–506. <https://doi.org/10.1007/S10994-021-05946-3>
- [17] Tian Jiang, Lin Tan, and Sunghun Kim. 2013. Personalized defect prediction. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 279–289. <https://doi.org/10.1109/ASE.2013.6693087>
- [18] Yue Jiang, Bojan Cukic, and Tim Menzies. 2008. Cost Curve Evaluation of Fault Prediction Models. *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, 197–206. <https://doi.org/10.1109/ISSRE.2008.54>
- [19] Jirayus Jirapakdee, Chakkrit Kla Tantithamthavorn, Hoa Khanh Dam, and John Grundy. 2022. An Empirical Study of Model-Agnostic Techniques for Defect Prediction Models. *IEEE Transactions on Software Engineering* 48, 1 (2022), 166–185. <https://doi.org/10.1109/TSE.2020.2982385>
- [20] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E. Hassan. 2016. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering* 21 (10 2016), 2072–2106. <https://doi.org/10.1007/s10664-015-9400-x>
- [21] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A Large-Scale Empirical Study of Just-in-Time Quality Assurance. *Software Engineering, IEEE Transactions on* 39 (06 2013), 757–773. <https://doi.org/10.1109/TSE.2012.70>
- [22] Taghi M. Khoshgoftaar, Xiaojing Yuan, Edward B. Allen, Wendell D. Jones, and John P. Hudepohl. 2002. Uncertain Classification of Fault-Prone Software Modules. *Empirical Software Engineering* 7 (2002), 297–318. <https://doi.org/10.1023/A:1020511004267>
- [23] Meelis Kull and Peter A. Flach. 2014. Reliability Maps: A Tool to Enhance Probability Estimates and Improve Classification Accuracy. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2014, Nancy, France, September 15-19, 2014. Proceedings, Part II (Lecture Notes in Computer Science, Vol. 8725)*, Toon Calders, Floriana Esposito, Eyke Hüllermeier, and Rosa Meo (Eds.). Springer, 18–33. https://doi.org/10.1007/978-3-662-44851-9_2
- [24] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead Jr. 2013. Does Bug Prediction Support Human Developers? Findings from a Google Case Study. In *Proceedings of the 2013 International Conference on Software Engineering (San Francisco, CA, USA) (ICSE '13)*. IEEE Press, 372–381.
- [25] Chenglin Li, Yangyang Zhao, and Yibiao Yang. 2022. An Empirical Study of the Bug Link Rate. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 177–188.
- [26] Gernot Liebchen and Martin Shepperd. 2016. Data Sets and Data Quality in Software Engineering: Eight Years On. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering (Ciudad Real, Spain) (PROMISE 2016)*. Association for Computing Machinery, New York, NY, USA, Article 7, 4 pages. <https://doi.org/10.1145/2972958.2972967>
- [27] Gernot A. Liebchen and Martin Shepperd. 2008. Data Sets and Data Quality in Software Engineering. In *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering (Leipzig, Germany) (PROMISE '08)*. Association for Computing Machinery, New York, NY, USA, 39–44. <https://doi.org/10.1145/1370788.1370799>
- [28] Mingxia Liu, Linsong Miao, and Daoqiang Zhang. 2014. Two-Stage Cost-Sensitive Learning for Software Defect Prediction. *IEEE Transactions on Reliability* 63, 2 (2014), 676–686. <https://doi.org/10.1109/TR.2014.2316951>
- [29] Shiran Liu, Zhaoqiang Guo, Yanhui Li, Chuanqi Wang, Lin Chen, Zhongbin Sun, Yuming Zhou, and Baowen Xu. 2022. Inconsistent defect labels: Essence, causes, and influence. *IEEE Transactions on Software Engineering* 49, 2 (2022), 586–610.
- [30] Huihua Lu, Ekrem Kocaguneli, and Bojan Cukic. 2014. Defect Prediction between Software Versions with Active Learning and Dimensionality Reduction. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 312–322. <https://doi.org/10.1109/ISSRE.2014.35>
- [31] Lech Madeyski and Marian Jureczko. 2014. Which process metrics can significantly improve defect prediction models? An empirical study. *Software Quality Journal* 23 (2014), 393 – 422. <https://api.semanticscholar.org/CorpusID:255071743>
- [32] Suvodeep Majumder, Pranav V. Mody, and Tim Menzies. 2020. Revisiting process versus product metrics: a large scale analysis. *Empirical Software Engineering* 27 (2020). <https://api.semanticscholar.org/CorpusID:221246191>
- [33] Shane McIntosh and Yasutaka Kamei. 2017. Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-In-Time Defect Prediction. *IEEE Transactions on Software Engineering* PP (04 2017), 1–1. <https://doi.org/10.1109/TSE.2017.2693980>
- [34] Thomas Melliush, Craig Saunders, Ilia Nouretdinov, and Volodya Vovk. 2001. Comparing the Bayes and Typicalness Frameworks. In *Machine Learning: ECML 2001, Luc De Raedt and Peter Flach (Eds.)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 360–371.
- [35] Diego P.P. Mesquita, Lincoln S. Rocha, João Paulo P. Gomes, and Ajalmar R. Rocha Neto. 2016. Classification with reject option for software defect prediction. *Applied Soft Computing* 49 (2016), 1085–1093. <https://doi.org/10.1016/j.asoc.2016.06.023>
- [36] Chao Ni, Wei Wang, Kaiwen Yang, Xin Xia, Kui Liu, and David Lo. 2022. The best of both worlds: integrating semantic features with expert features for defect prediction and localization. 672–683. <https://doi.org/10.1145/3540250.3549165>

⁷<https://zenodo.org/records/3965246#.XyEDVnUzY5k>

⁸<https://github.com/Anonym0u5Submi55i0n/CDP>

- [37] Chao Ni, Xiaodan Xu, Kaiwen Yang, and David Lo. 2023. Boosting Just-in-Time Defect Prediction with Specific Features of C/C++ Programming Languages in Code Changes. In *20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023*. IEEE, 472–484. <https://doi.org/10.1109/MSR59073.2023.00072>
- [38] Chao Ni, Xiaodan Xu, Kaiwen Yang, and David Lo. 2023. Boosting Just-in-Time Defect Prediction with Specific Features of C/C++ Programming Languages in Code Changes. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 472–484. <https://doi.org/10.1109/MSR59073.2023.00072>
- [39] Chao Ni, Kaiwen Yang, Yan Zhu, Xiang Chen, and Xiaohu Yang. 2023. Unifying Defect Prediction, Categorization, and Repair by Multi-Task Deep Learning. 1980–1992. <https://doi.org/10.1109/ASE56229.2023.00083>
- [40] Liang Niu, Jianwu Wan, Hongyuan Wang, and Kaiwei Zhou. 2020. Cost-Sensitive Dictionary Learning for Software Defect Prediction. *Neural Process. Lett.* 52, 3 (dec 2020), 2415–2449. <https://doi.org/10.1007/s11063-020-10355-z>
- [41] Sushant Kumar Pandey, Ravi Bhushan Mishra, and Anil Kumar Tripathi. 2021. Machine learning based methods for software fault prediction: A survey. *Expert Systems with Applications* 172 (2021), 114595. <https://doi.org/10.1016/j.eswa.2021.114595>
- [42] Harris Papadopoulos, Kostas Proedrou, Volodya Vovk, and Alex Gammerman. 2002. Inductive Confidence Machines for Regression. In *Machine Learning: ECML 2002*, Tapio Elomaa, Heikki Mannila, and Hannu Toivonen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 345–356.
- [43] Harris Papadopoulos, Volodya Vovk, and Alex Gammerman. 2007. Conformal Prediction with Neural Networks. In *19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2007), October 29-31, 2007, Patras, Greece, Volume 2*. IEEE Computer Society, 388–395. <https://doi.org/10.1109/ICTAI2007.47>
- [44] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2019. Fine-grained just-in-time defect prediction. *J. Syst. Softw.* 150 (2019), 22–36. <https://doi.org/10.1016/j.jss.2018.12.001>
- [45] Chanathip Pornprasit and Chakrit Kla Tantithamthavorn. 2021. JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction. *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR) (2021)*, 369–379. <https://api.semanticscholar.org/CorpusID:232223034>
- [46] Chanathip Pornprasit and Chakrit Kla Tantithamthavorn. 2023. DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction. *IEEE Transactions on Software Engineering* 49, 1 (2023), 84–98. <https://doi.org/10.1109/TSE.2022.3144348>
- [47] Sophia Quach, Maxime Lamothe, Bram Adams, Yasutaka Kamei, and Weiyi Shang. 2021. Evaluating the Impact of Falsely Detected Performance Bug-Inducing Changes in JIT Models. *Empirical Softw. Engg.* 26, 5 (sep 2021), 32 pages. <https://doi.org/10.1007/s10664-021-10004-6>
- [48] Foyzur Rahman and Premkumar Devanbu. 2013. How, and why, process metrics are better. In *2013 35th International Conference on Software Engineering (ICSE)*. 432–441. <https://doi.org/10.1109/ICSE.2013.6606589>
- [49] Santosh Rathore and Sandeep Kumar. 2021. An empirical study of ensemble techniques for software fault prediction. *Applied Intelligence* 51 (06 2021), 1–30. <https://doi.org/10.1007/s10489-020-01935-6>
- [50] Glenn Shafer and Vladimir Vovk. 2008. A Tutorial on Conformal Prediction. *J. Mach. Learn. Res.* 9 (jun 2008), 371–421.
- [51] Xhulja Shahini, Domenic Bubel, and Andreas Metzger. 2023. Variance of ML-based software fault predictors: are we really improving fault prediction? *CoRR abs/2310.17264* (2023). <https://doi.org/10.48550/ARXIV.2310.17264> arXiv:2310.17264
- [52] Mohammad Hossein Shaker and Eyke Hüllermeier. 2020. Aleatoric and Epistemic Uncertainty with Random Forests. In *Advances in Intelligent Data Analysis XVIII - 18th International Symposium on Intelligent Data Analysis, IDA 2020, Konstanz, Germany, April 27-29, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12080)*, Michael R. Berthold, Ad Feelders, and Georg Krempel (Eds.). Springer, 444–456. https://doi.org/10.1007/978-3-030-44584-3_35
- [53] Martin Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair. 2013. Data Quality: Some Comments on the NASA Software Defect Datasets. *Software Engineering, IEEE Transactions on* 39 (09 2013), 1208–1215. <https://doi.org/10.1109/TSE.2013.11>
- [54] Jacek Sliwinski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM SIGSOFT Softw. Eng. Notes* 30, 4 (2005), 1–5. <https://doi.org/10.1145/1082983.1083147>
- [55] Sadia Tabassum, Leandro L. Minku, Danyi Feng, George G. Cabral, and Liyan Song. 2020. An Investigation of Cross-Project Learning in Online Just-in-Time Software Defect Prediction. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (Seoul, South Korea) (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 554–565. <https://doi.org/10.1145/3377811.3380403>
- [56] Vladimir Vovk. 2013. Conditional validity of inductive conformal predictors. *Mach. Learn.* 92, 2-3 (2013), 349–376. <https://doi.org/10.1007/s10994-013-5355-6>
- [57] Volodya Vovk, Alexander Gammerman, and Craig Saunders. 1999. Machine-Learning Applications of Algorithmic Randomness. In *Proceedings of the Sixteenth International Conference on Machine Learning (ICML 1999), Bled, Slovenia, June 27 - 30, 1999*, Ivan Bratko and Saso Dzeroski (Eds.). Morgan Kaufmann, 444–453.
- [58] Romi Wahono. 2015. A Systematic Literature Review of Software Defect Prediction: Research Trends, Datasets, Methods and Frameworks. *Journal of Software Engineering* 1 (05 2015).
- [59] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically Learning Semantic Features for Defect Prediction. In *Proceedings of the 38th International Conference on Software Engineering (Austin, Texas) (ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 297–308. <https://doi.org/10.1145/2884781.2884804>
- [60] Xinli Yang, David Lo, Xin Xia, and Jianling Sun. 2017. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology* 87 (03 2017). <https://doi.org/10.1016/j.infsof.2017.03.007>
- [61] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep Learning for Just-in-Time Defect Prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, Vancouver, BC, Canada, August 3-5, 2015*. IEEE, 17–26. <https://doi.org/10.1109/QRS.2015.14>
- [62] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 157–168. <https://doi.org/10.1145/2950290.2950353>
- [63] Ruijie Chen Xiaolin Ju Yubin Qu, Xiang Chen and Jiangfeng Guo. 2019. Active Learning using Uncertainty Sampling and Query-by-Committee for Software Defect Prediction. *International Journal of Performability Engineering* 15, 10, Article 2701 (2019), 7 pages. <https://doi.org/10.23940/ijpe.19.10.p16.27012708>
- [64] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. 2021. Deep just-in-time defect prediction: how far are we?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (Virtual, Denmark) (ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 427–438. <https://doi.org/10.1145/3460319.3464819>
- [65] Huan Zhang, Li Kuang, Aolang Wu, Qiuning Zhao, and Xiaoxian Yang. 2023. Just-in-time defect prediction enhanced by the joint method of line label fusion and file filtering. *IET Software* 17 (06 2023), n/a–n/a. <https://doi.org/10.1049/sfw2.12131>
- [66] Tanghaoran Zhang, Yue Yu, Xinjun Mao, Yao Lu, Zhixing Li, and Huaimin Wang. 2022. FENSE: A Feature-Based Ensemble Modeling Approach to Cross-Project Just-in-Time Defect Prediction. *Empirical Softw. Engg.* 27, 7 (dec 2022), 41 pages. <https://doi.org/10.1007/s10664-022-10185-8>
- [67] Yunhua Zhao, Kostadin Damevski, and Hui Chen. 2023. A Systematic Survey of Just-in-Time Software Defect Prediction. *ACM Comput. Surv.* 55, 10, Article 201 (feb 2023), 35 pages. <https://doi.org/10.1145/3567550>
- [68] Xin Zhou, Bowen Xu, DongGyun Han, Zhou Yang, Junda He, and David Lo. 2023. CCBERT: Self-Supervised Code Change Representation Learning. *CoRR abs/2309.15474* (2023). <https://doi.org/10.48550/ARXIV.2309.15474> arXiv:2309.15474
- [69] Ömer Faruk Arar and Kürşat Ayan. 2015. Software defect prediction using cost-sensitive neural network. *Applied Soft Computing* 33 (2015), 263–277. <https://doi.org/10.1016/j.asoc.2015.04.045>

Received November 2023; December 2023; January 2024