# JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction

Chanathip Pornprasit, Chakkrit (Kla) Tantithamthavorn
Monash University, Melbourne, Australia.

*Abstract*—A Just-In-Time (JIT) defect prediction model is a classifier to predict if a commit is defect-introducing. Recently, CC2Vec—a deep learning approach for Just-In-Time defect prediction—has been proposed. However, CC2Vec requires the whole dataset (i.e., training + testing) for model training, assuming that all unlabelled testing datasets would be available beforehand, which does not follow the key principles of just-in-time defect predictions. Our replication study shows that, after excluding the testing dataset for model training, the F-measure of CC2Vec is decreased by 38.5% for OpenStack and 45.7% for Qt, highlighting the negative impact of excluding the testing dataset for Just-In-Time defect prediction. In addition, CC2Vec cannot perform fine-grained predictions at the line level (i.e., which lines are most risky for a given commit).

In this paper, we propose JITLine—a Just-In-Time defect prediction approach for predicting defect-introducing commits and identifying lines that are associated with that defect-introducing commit (i.e., defective lines). Through a case study of 37,524 commits from OpenStack and Qt, we find that our JITLine approach is at least 26%-38% more accurate (F-measure), 17%-51% more cost-effective (PCI@20%LOC), 70-100 times faster than the state-of-the-art approaches (i.e., CC2Vec and DeepJIT) and the fine-grained predictions at the line level by our approach are 133%-150% more accurate (Top-10 Accuracy) than the baseline NLP approach. Therefore, our JITLine approach may help practitioners to better prioritize defect-introducing commits and better identify defective lines.

## I. INTRODUCTION

Modern software development cycles tend to release software products in a short-term period. Such short-term software development cycles often pose critical challenges to modern Software Quality Assurance (SQA) practices. Therefore, continuous code quality tools (e.g., CI/CD, modern code review, static analysis) have been heavily adopted to early detect software defects. However, SQA teams cannot effectively inspect every commit given limited SQA resources.

Just-in-time (JIT) defect prediction [16, 19] is proposed to predict if a commit will introduce defects in the future. Such commit-level predictions are useful to help practitioners prioritize their limited SQA resources on the most risky commits during the software development process. In the past decades, several machine learning approaches are employed for developing JIT defect prediction models [7, 18, 29, 32]. However, these approaches often rely on handcrafted commit-level features (e.g., Churn).

Recently, several deep learning approaches have been proposed for Just-In-Time defect prediction (e.g., DeepJIT [9] and CC2Vec [10]). Hoang *et al.* [10] found that their CC2Vec approach outperforms DeepJIT for Just-In-Time defect pre-

diction. CC2Vec requires both training and unlabelled testing datasets for training CC2Vec models, assuming that all unlabelled testing datasets would be available beforehand. However, these assumptions of CC2Vec do not follow the key principles of the Just-In-Time defect prediction: (1) the predictions of the CC2Vec approach cannot be made immediately for a newly arrived commit; and (2) it is unlikely that the unlabelled testing dataset would be available beforehand when training JIT models. Thus, we perform a replication study to confirm the merit of previous experimental findings and extend their experiment by excluding testing datasets and evaluate with five additional evaluation measures.

**RS1) Can we replicate the results of deep learning approaches for Just-In-Time defect prediction?**
Similar to the original study [10], we are able to replicate the results of CC2Vec.

**RS2) How does CC2Vec perform for Just-In-Time defect prediction after excluding testing datasets?**
After excluding testing datasets when developing the JIT models, we find that the F-measure of CC2Vec is decreased by 38.5% for OpenStack and 45.7% for Qt. In addition, CC2Vec achieves a high False Alarm Rate (FAR) of 0.87 for OpenStack and 0.63 for Qt, indicating that 63%-87% clean commits are incorrectly predicted as defect-introducing. Thus, developers still waste many unnecessarily effort to inspect clean commits that are incorrectly predicted as defect-introducing.

In addition, Hoang *et al.* [10] did not compare their approach with simple JIT approaches, did not evaluate the cost-effectiveness, did not report the computational time, and cannot perform fine-grained predictions at the line level. Thus, it remains unclear about the practical value of the CC2Vec approach when considering the amount of effort that developers need to inspect.

In this paper, we propose JITLine—a machine learning-based Just-In-Time defect prediction approach that can both predict defect-introducing commits and identify defective lines that are associated with that commit. We evaluate our JITLine approach with the state-of-the-art commit-level JIT defect prediction approaches (i.e., EARL [17], DeepJIT [9], and CC2Vec [10]) with respect to six traditional measures (i.e, AUC, F-measure, False Alarm Rate, Distance-to-Heaven, Precision, and Recall), three cost-effectiveness measures (i.e., PCI@20%LOC, Effort@20%Recall, $P_{Opt}$). In addition, we also compare our approach with a baseline line-level JIT

defect localization by Yan [43] using four line-level effort-aware measures (i.e., Top-10 Accuracy, Recall@20%LOC, Effort@20%Recall$_{line}$, Initial False Alarm). Through a case study of 37,524 total commits that span across two large-scale open-source software projects (i.e., OpenStack and Qt), we address the following four research questions:

**RQ1)** **Does our JITLine <u>outperform</u> the state-of-the-art JIT defect prediction approaches?**
Our JITLine approach achieves F-measure 26%-38% higher than the state-of-the-art approaches (i.e., CC2Vec). Our JITLine achieves a False Alarm Rate (FAR) 94%-97% lower than the CC2Vec approach.

**RQ2)** **Is our JITLine more <u>cost-effective</u> than the state-of-the-art JIT defect prediction approaches?**
Our JITLine is 17%-51% more cost-effective than the state-of-the-art approaches in term of PCI20%Effort. In addition, our JITLine can save the amount of effort by 89%-96% to find the same number of actual defect-introducing commits (i.e., 20% Recall) when compared to the state-of-the-art approaches.

**RQ3)** **Is our JITLine <u>faster</u> than the state-of-the-art JIT defect prediction approaches?**
Our JITLine is 70-100 times faster than the deep learning approaches for Just-In-Time defect prediction.

**RQ4)** **How effective is our JITLine for prioritizing defective <u>lines</u> of a given defect-introducing commit?**
Our JITLine approach is 133%-150% more accurate than the baseline approach by Yan *et al.* [43] for identifying actual defective lines in the top-10 recommendations. Our JITLine approach requires 17%-27% less amount of effort than the baseline approach in order to find the same amount of actual defective lines.

<u>Contributions</u>. The contributions of this paper are as follows:

- We conduct a replication study of the state-of-the-art deep learning approach (CC2Vec [10]) for JIT defect prediction and extend their experiment by excluding testing datasets with five evaluation measures (Section III).
- We propose JITLine—a machine learning-based Just-In-Time defect prediction approach that can both predict defect-introducing commits and identify their associated defective lines (Section V).
- We evaluate our JITLine approach at the commit level with the state-of-the-art JIT defect prediction approaches with respect to six traditional measures, three cost-effectiveness measures, and at the line level with four effort-aware line-level measures (Section VI).
- Our results show that our JITLine approach outperforms (RQ1), more cost-effective (RQ2), faster (RQ3), and more fine-grained (RQ4) than the state-of-the-art approaches.

## II. BACKGROUND

Commits created by developers are often used to describe new features, bug fixes, refactoring, etc. One commit contains three main pieces of information, i.e., a commit message, a code change, and their meta-data information (e.g., churn, author name). The commit message is used to describe the semantics of the code changes, while the code change indicates changed lines (i.e., added/modified/deleted lines).

In large-scale software projects, there is a stream of commits that developers need to review and inspect. However, due to the limited SQA resources, Just-In-Time defect prediction approaches have been proposed to help developers prioritize their limited SQA resources on the most risky commits [17, 18]. Below, we discuss three state-of-the-art approaches for Just-In-Time defect prediction.

*EALR* [17] is an Effort-Aware JIT defect prediction method using a Logistic Regression model with traditional commit-level software metrics (e.g., churn). EALR generates a rank of defect-introducing commits by considering the amount of inspection effort—i.e., the predicted probability is normalized by the commit size (i.e., churn). However, such techniques often rely on handcrafted feature engineering.

*DeepJIT* [9] is an end-to-end deep learning framework for Just-in-Time defect prediction. DeepJIT automatically generates features using a Convolutional Neural Network (CNN) architecture. Generally, DeepJIT takes the commit message and the code change as an input into two CNN models in order to generate a vector representation—i.e, one CNN for generating commit message vectors and another CNN for generating code changes vectors. Finally, the concatenation of both the commit message vector and the code change vector is input into the fully-connected layer to generate the probability of defect-introducing commit.

*CC2Vec* [10] is an approach to learn the distributed representation of commit. Traditionally, one commit has a hierarchical structure–i.e., one commit consists of changed files, one change file consists of changed hunks, one change hunk consists of changed lines, one changed line consists of changed tokens. Unlike DeepJIT that ignores the information about the hierarchical structure of code commits, CC2Vec has been proposed to automatically learn the hierarchical structure of code commits using a Hierarchical Attention Network (HAN) architecture. The goal of CC2Vec is to learn the relationship between the actual code changes and the semantic of that code changes (i.e., the first line of commit messages). Then, in the feature extraction layer, HAN is used to build vector representations of changed lines; these vectors are then used to construct vector representations of hunks; and then these vectors are aggregated to construct the embedding vector of the removed or added code. Then, the embedding vectors of the removed code and added code is input into a fully-connected layer to generate a vector that represents the code change.

Recently, Hoang *et al.* has shown that the combination of CC2Vec and DeepJIT outperforms the stand-alone DeepJIT approach. In particular, they used CC2Vec to generate a vector representation of code changes. Then, such code changes vector is concatenated with the commit message vectors and the code change vectors that are generated by DeepJIT to generate a final vector representation. Finally, the concatenation vector is input into the fully-connected layer to generate the probability of defect-introducing commit.

TABLE I: The results of our replication study of CC2Vec [10] when using "train+test" and "train only" for model training.

| | | OpenStack | | | | | | Qt | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | AUC | F1 | FAR | d2h | Precision | Recall | AUC | F1 | FAR | d2h | Precision | Recall |
| CC2Vec [Train+Test] | Original | 0.81 | - | - | - | - | - | 0.82 | - | - | - | - | - |
| | Ours | 0.80 | 0.39 | 0.26 | 0.28 | 0.27 | 0.70 | 0.84 | 0.35 | 0.17 | 0.25 | 0.24 | 0.70 |
| CC2Vec [Train Only] | Ours | 0.77 | 0.24 | 0.87 | 0.61 | 0.14 | 0.99 | 0.81 | 0.19 | 0.63 | 0.45 | 0.10 | 0.96 |

## III. A Replication Study of the State-of-the-art Deep Learning Approach for JIT Defect Prediction

In this section, we present the motivation, approach, and results of our replication study (RS) of CC2Vec for Just-In-Time defect prediction.

**Motivation.** One of the key principles of *Just-In-Time* defect prediction models is to *generate predictions as soon as possible* for a newly arrived commit. Let's consider $T_1$ as the present (see Figure 1), the whole historical data will be used for training a JIT model in order to immediately generate a prediction of a newly arrived commit. However, CC2Vec requires both training and unlabelled testing datasets for training CC2Vec models (i.e., the periods of $T_0$-$T_1$ and $T_1$-$T_2$), assuming that all unlabelled testing datasets would be available beforehand. In particular, Hoang *et al.* (Section 3.3.3 of the original study [10]) stated that *"CC2Vec is first used to learn distributed representations of the code changes in the whole dataset. All patches from the training and testing dataset are used since the log messages of the testing dataset are not part of the predictions of the task"*. This indicates that the unlabelled testing dataset needs to be available beforehand for training CC2Vec models. However, these assumptions of CC2Vec do not follow the key principles of the Just-In-Time defect prediction: (1) the predictions of the CC2Vec approach cannot be made immediately for a newly arrived commit; and (2) it is unlikely that the unlabelled testing dataset would be available beforehand when training JIT models. Thus, it remains unclear what the performance of CC2Vec for Just-In-Time defect prediction is after considering the key principle of Just-In-Time defect prediction (i.e., excluding testing dataset for model training). In addition, several other performance measures (e.g., F-measure) have not been evaluated in the original study. Thus, we (RS1) perform a replication study to confirm the merit of previous experimental findings and (RS2) extend their experiment by excluding testing datasets and evaluate with five additional evaluation measures.

### (RS1) Can we replicate the results of deep learning approaches for Just-In-Time defect prediction?

<u>Approach.</u> To address RS1, we first download the replication package of Hoang *et al.* [10]. We carefully study the replication package to understand all details. Then, we execute the source code followed by the instructions and datasets provided by Hoang *et al.* [10]. Finally, we compute a relative percentage between our results and the original paper as follows: $\% = \left(\frac{\text{ours} - \text{original}}{\text{original}}\right) \times 100\%$

<u>Results.</u> **Similar to the original study [10], we are able to replicate the results of CC2Vec.** Table I (see the green cells) shows that, in our experiment, CC2Vec achieves an AUC



Fig. 1: The comparison between the workflow of JITLine that can immediately generate predictions and the workflow of CC2Vec+DeepJIT [10] which requires testing dataset to be available beforehand for training CC2Vec+DeepJIT models.

of 0.80 for OpenStack and 0.84 for Qt, while the original paper reported an AUC of 0.81 for OpenStack and 0.82 for Qt. Our results are only 1%-2% different when compared to the original paper. This finding confirms that the results of CC2Vec are replicable for Just-In-Time defect prediction.

### (RS2) How does CC2Vec perform for Just-In-Time defect prediction after excluding testing datasets?

<u>Approach.</u> To address RS2, we repeat the experiment of Hoang *et al.* [10] in two settings—i.e., the original experiment with training and testing datasets and our experiment with training datasets only. In addition, we extend their experiment by evaluating the CC2Vec approach using five additional evaluation measures (i.e., F-measure, False Alarm Rate, Distance-to-Heaven, Precision, and Recall).

<u>Results.</u> **After excluding testing datasets when developing the JIT models, we find that the F-measure of CC2Vec is decreased by 38.5%(0.35→0.19) for OpenStack. and 45.7%(0.39→0.24) for Qt.** Table I (see the red cells) shows the result between two experimental settings (i.e., training+testing vs. training only) with respect to AUC, F1, FAR, d2h, Precision and Recall. We find that the values of several performance measures (i.e, AUC, F-measure, FAR, d2h) are negatively impacted by the exclusion of the testing datasets. We find that AUC is decreased by 3.9% for OpenStack and 3.7% for Qt, while False Alarm Rates (FAR) are increased by 234.62% for OpenStack and 270.59% for Qt. Similarly, the d2h value is increased by 126.92% for Openstack and 80% for Qt. The higher FAR and d2h of CC2Vec has to do with the substantial increasing Recall to 0.99 for OpenStack and 0.96

for Qt—i.e., CC2Vec predicts most of the commits as defect-introducing (higher Recall), but many of the predictions are incorrect (higher FAR, less Precision). These findings indicate that the exclusion of testing datasets in model training has a large negative impact on the performance of CC2Vec (i.e., producing higher False Alarm Rates). Thus, developers have to waste unnecessarily effort on inspecting clean commits that are incorrectly predicted as defect-introducing.

## IV. RELATED WORK AND RESEARCH QUESTIONS

In this section, we discuss the following four main limitations of prior studies with respect to the literature in order to motivate our approach and research questions.

**First, several traditional machine learning-based JIT approaches have not been compared with the deep learning approaches for JIT defect prediction.** Recently, researchers found that several simple approaches often outperform deep learning approaches in SE tasks. For example, Hellendoorn [8], Fu and Menzies [5], Liu *et al.* [22]. Menzies *et al.* [25] suggested that researchers should explore simple and fast approaches before applying deep learning approaches on SE tasks. However, Hoang *et al.* [10] did not compared their CC2Vec approach with other simple approaches (e.g., logistic regression and random forest). Therefore, we wish to investigate if our approach outperforms the deep learning approaches for Just-In-Time defect prediction.

**Second, the cost-effectiveness of deep learning approaches for JIT defect prediction has not been investigated.** Prior work pointed out that different code changes often require different amount of code inspection effort [11, 24]—i.e., large code changes often require a high amount of code inspection effort. However, Hoang *et al.* [10] did not investigate the cost-effectiveness of their CC2Vec approach. In addition, the CC2Vec approach does not take into consideration the effort required to inspect code changes when prioritizing defect-introducing commits. Therefore, we wish to investigate if our approach is more cost-effective than the deep learning approaches for Just-In-Time defect prediction.

**Third, the computational time of deep learning approaches JIT defect prediction has not been investigated.** Several researchers raised concerns that deep learning approaches are often complex and very expensive in terms of GPU costs/CPU time. For example, Jiang *et al.* [12]'s approach requires 38 hours for training their deep learning models on NVIDIA GeForce GTX 1070. Menzies *et al.* [25] found that a simple approach that is 500+ times faster achieves similar performance to deep learning approaches. Therefore, we wish to investigate if our approach is faster than the deep learning approaches for Just-In-Time defect prediction.

**Finally, there exists no machine learning approaches for fine-grained Just-In-Time defect prediction at line level.** Recently, Pascarella *et al.* [26] proposed a fine-grained JIT defect prediction model which based on handcrafted features to prioritize which changed files in a commit should be review first. However, this approach cannot identify defective lines of the changed files. Recently, Yan *et al.* [43] proposed a

fine-grained JIT defect localization at the line level to help developers to locate and address defects using less effort. Yan *et al.* [43] proposed a two-phase approach—i.e., the ML model trained on software metrics (e.g., #added_lines) is first used to identify which commits are the most risky, then the N-gram model trained on textual features is finally used to localise the riskiest lines. On the other hand, a recent work by Wattanakriengkrai *et al.* [42] pointed out that a machine learning approach outperforms the n-gram approach. However, their experiment focused solely on file-level defect prediction—not Just-In-Time defect prediction. Therefore, we wish to investigate if our approach is more effective than the two-phase approach for Just-In-Time defect prediction.

Considering the limitations yet high impact of prior work, we propose JITLine—a machine learning-based Just-In-Time defect prediction approach that can predict both defect-introducing commits and their associated defective lines. Then, we formulate the following research questions:

RQ1) Does our JITLine <u>outperform</u> the state-of-the-art JIT defect prediction approaches?

RQ2) Is our JITLine more <u>cost-effective</u> than the state-of-the-art JIT defect prediction approaches?

RQ3) Is our JITLine <u>faster</u> than the state-of-the-art JIT defect prediction approaches?

RQ4) How effective is our JITLine for prioritizing defective <u>lines</u> of a given defect-introducing commit?

## V. JITLINE: A JIT DEFECT PREDICTION APPROACH AT THE COMMIT AND LINE LEVELS

In this section, we present the implementation of our JITLine approach. The goal of our JITLine approach is to predict defect-introducing commits and identify lines that are associated with that defect-introducing commit (i.e., defective lines). The underlying intuition of our approach is that code tokens that frequently appeared in defect-introducing commits in the past are likely to be fixed in the future.

**<u>Overview.</u>** Our approach begins with extracting source code tokens of code changes as features (i.e., token features). Since our JIT defect datasets are highly imbalanced (i.e., 8%-13% defective ratio), we apply a SMOTE technique that is optimized by a Differential Evolution (DE) algorithm to handle the class imbalance issue on a training dataset. Then, we build commit-level JIT defect prediction model using the rebalanced training dataset. Next, we generate a prediction for each commit in a testing dataset. After that, we normalize the prediction score by the amount of code changes (i.e., churn) in order to consider the inspection effort when generating the ranking of defect-introducing commits. For each commit in the testing dataset, we extract the importance score of each token features using a state-of-the-art model-agnostic technique, i.e., Local Interpretable Model-Agnostic Explanations (LIME). Finally, we rank defective lines that are associated with a given commit based on the LIME's importance scores. We describe each step in details below.

**(Step 1) Extracting Bag-of-Tokens Features.** Following the underlying intuition of our approach, we represent each

commit using Bag-of-Tokens features (i.e., the frequency of each code token in a commit). To do so, for each commit, we first perform a code tokenization step to break each changed line into separate tokens. Then, we parse its removed lines or added lines into a sequence of tokens. As suggested by Rahman *et al.* [27], removing these non-alphanumeric characters will ensure that the analyzed code tokens will not be artificially repetitive. Thus, we apply a set of regular expressions to remove non-alphanumeric characters such as semi-colon (;) and equal sign (=). We also replace the numeric literal and string literal with a special token (i.e., $<$NUM$>$ and $<$STR$>$ respectively) to reduce the vocabulary size. Then, we extract the frequency of code tokens for each commit using the `Countvectorize` function of the Scikit-Learn Python library. We neither perform lowercase, stemming, nor lemmatization (i.e., a technique to reduce inflectional forms) on our extracted tokens, since the programming language of our studied systems is case-sensitive. Otherwise, the meaning of code tokens may be discarded if stemming and lemmatization are applied.

**(Step 2) Handling class imbalance using an Optimized SMOTE technique.** Since our JIT defect datasets are highly imbalanced (i.e., 8%-13% defective ratio), we apply a SMOTE technique that is optimized by a Differential Evolution (DE) algorithm to handle the class imbalance issue on a training dataset. The training dataset is splitted into a new training set and a validation set. The new training set is used to train DE+SMOTE, while the validation set is used to select the best hyper-parameter settings. We select the SMOTE technique, as prior studies have shown that the SMOTE technique outperforms other class rebalancing techniques [2, 37].

The SMOTE technique starts with a set of minority class (i.e., defect-introducing commits). For each of the minority class of the training datasets, SMOTE calculates the $k$-nearest neighbors. Then, SMOTE selects $N$ instances of the majority class (i.e., clean commits) based on the smallest magnitude of the euclidean distances that are obtained from the k-nearest neighbors. Finally, SMOTE combines the synthetic oversampling of the minority defect-introducing commits with the undersampling the majority clean commits. We use the implementation of `SMOTE` function provided by the `Imbalanced-Learn` Python library [20]. However, prior studies pointed out that the SMOTE technique involves many parameters settings (e.g., $k$ the number of neighbors, $m$ the number of synthetic examples to create, $r$ the power parameter for the Minkowski distance metric), which often impact the accuracy of prediction models [2, 6, 37, 39].

To ensure that we achieve the best performance of the SMOTE algorithm, we optimize the SMOTE technique using a Differential Evolution (DE) algorithm (as suggested by Agrawal *et al.* [2] and Tantithamthavorn *et al.* [37]). DE [35] is an evolutionary-based optimization technique, which is based on a differential equation concept. Unlike a Genetic Algorithm technique that uses crossover as search mechanisms, a DE technique uses mutation as a search mechanism. First, DE generates an initial population of candidate setting of SMOTE's

$k$ nearest neighbors with a range value of 1-20. Then, DE generates new candidates by adding a weighted difference between two population members to the third member based on a crossover probability parameter. Finally, DE keeps the best candidate SMOTE's parameter setting that is evaluated by a fitness function of maximizing an AUC value for the next generation. We use the implementation of the differential evolution algorithm provided by Scipy Python library [41]. As suggested by Agrawal *et al.* [2], we set the population size to 10, the mutation power to 0.7 and a crossover probability (or `recombination` parameter in Scipy) to 0.3.

**(Step 3) Building commit-level JIT defect prediction models.** We build a commit-level JIT defect model using both the Bag-of-Tokens features from Step 1 and the commit-level metrics from McIntosh and Kamei [23]. The details of commit-level metrics are provided in the replication package. Prior work found that different classification techniques often produce different performance measures. Thus, we conduct an experiment on different classification techniques. We consider the following well-known classification techniques [1, 2, 39, 40], i.e., Random Forest (RF), Logistic Regression (LR), Support Vector Machine (SVM), k-Nearest Neighbours (kNN), and AdaBoost. For each project, we build the JIT model using the implementation provided by Python Scikit-Learn package. We find that LR, kNN, and SVM cannot be built with the Qt project due to the high-dimensional feature space, and the model training time for such models (which takes few hours) is considerably larger than RF (which takes few minutes). Therefore, we only select the Random Forest classification technique for our study. After we experiment with different parameter settings of trees (a range of 50 to 1,000), we find that our approach is not sensitive to the parameter setting of random forest. Thus, we set the number of tress of random forest to 300.

**(Step 4) Computing a defect density of each commit.** We then generate the prediction probability for each commit in the testing dataset using the `predict_proba` function provided by the Scikit-learn Python library. Then, we compute the defect density as the probability score normalized by the total changed lines of code of that commit ($\frac{\text{Y}(m)}{\#\text{LOC}(c)}$). The use of defect density is suggested by prior studies [17, 24] who argued that the cost of applying quality assurance activities may not be the same for each code changes. In other words, a prediction model that prioritizes the largest commit as most defect prone would have a very high recall, i.e., those commits likely contain the majority of defects, yet inspecting all those commits would take a considerable amount of time. In contrast, a model that recommends slightly less defect-prone commits that are smaller to inspect would be more cost-effective [17].

**(Step 5) Generating a ranking of defective lines for a given commit.** In our studied projects, we found that the average size of the commit varies from 73 to 140 changed lines, but the average ratio of actual defective lines is as low as 51%-53%. Thus, developers still spend unnecessarily effort on locating actual defective lines of that commit [42]. To address

TABLE II: The statistics of our studied datasets.

| | #Commits | %Defect-Introducing Commits | #Unique Tokens | Avg. of Commit Size | Avg. of %Defective Lines |
|---|---|---|---|---|---|
| Openstack | 12,374 | 13% | 32K | 73 LOC | 53% |
| Qt | 25,150 | 8% | 81K | 140 LOC | 51% |

this challenge, we propose to generate a ranking of defective lines for a given commit. For each commit, we compute the importance score of token features using a Local Interpretable Model-agnostic Explanations (LIME) technique. LIME [30] is a model-agnostic technique that aims to mimic the behavior of the predictions of the defect model by explaining the individual predictions. Given a commit-level JIT defect prediction model and a commit in the testing dataset, LIME performs the following steps:

1) Generate neighbor instances of a test instance $x$. LIME randomly generates $n$ synthetic instances surrounding the test instance $x$ using a random perturbation method with an exponential kernel function on cosine distance.
2) Generate labels of the neighbors using a commit-level JIT defect prediction model. LIME uses the commit-level JIT defect prediction model to generate the predictions of the neighbor instances.
3) Generates local explanations from the generated neighbors. LIME builds a local sparse linear regression model (K-Lasso) using the randomly generated instances and their generated predictions from the commit-level defect model. The coefficients of the K-Lasso model indicate the importance score of each feature on the prediction of a test instance according to the K-Lasso model.

The LIME's importance score of each token feature ranges from -1 to 1. A positive LIME score of a token feature ($0 < e \leq 1$) indicates that the feature has a positive impact on the estimated probability of the test instance (i.e., **risky tokens**). On the other hand, a negative LIME score of a token feature ($-1 \leq e < 0$) indicates that the token feature has a negative impact on the estimated probability (i.e., **non-risky tokens**). Once the importance score of each token is computed, we generate the ranking of defect-prone lines using the summation of the importance score for all tokens that appear in that line. We use the implementation of LIME provided by the `lime` Python package.

## VI. EXPERIMENTAL SETUP AND RESULTS

In this section, we describe the studied datasets and present the experimental results with respect to our four research questions.

**Studied Datasets.** In this paper, we select the dataset of McIntosh and Kamei [23] due to the following reasons. First, we would like to establish a fair comparison, using the same training and testing datasets with previous work [9, 10], where this dataset was used. Second, we would like to ensure that our results rely on high quality datasets. Recently, researchers raised concerns that the SZZ algorithm [33] may produce many false positives and false negatives [31]. However, the

datasets of McIntosh and Kamei [23] have been manually verified through many filtering steps (e.g., ignore comment updates, ignore white space/indentation changes, remove mislabelled defect-introducing commits). Finally, we select the datasets of McIntosh and Kamei [23] with two open-source software systems, i.e., OpenStack and Qt. Openstack is an opensource software for cloud infrastructure service. Qt is a cross-platform application development framework written in C++. Table II presents the statistics of the studied datasets.

Below, we present the approach and the results with respect to our four research questions.

### (RQ1) Does our JITLine _outperform the state-of-the-art JIT defect prediction approaches?_

Approach. To answer this RQ, we evaluate our JITLine using the same training/testing datasets as prior studies [9, 10, 23] to establish a fair comparison. For training, we use 11,043 commits for OpenStack and 22,579 commits for Qt. For testing, we use 1,331 commits for OpenStack and 2,571 for Qt. Since our JIT defect datasets are time-wise, we do not perform cross-validation to avoid the use of testing data in the training data [15]. Then, we compare our JITLine with the following three state-of-the-art JIT defect prediction approaches (i.e., EARL, DeepJIT, CC2Vec). The details of the state-of-the-art approach is provided in Section II. Finally, we evaluate these approaches using the following six traditional evaluation measures [2, 9, 10, 42].

1) AUC is an Area Under the ROC Curve (i.e., the true positive rate and the false positive rate). AUC values range from 0 to 1, with a value of 1 indicates perfect discrimination, while a value of 0.5 indicates random guessing.
2) F-measure is a harmonic mean of precision and recall, which is computed as $\frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$. We use the probability threshold of 0.5 for calculating precision and recall.
3) False Alarm Rate (FAR) [1] measures the ratio of incorrectly predicted defect-introducing commits and the number of actual clean commits $\frac{\text{FP}}{\text{FP} + \text{TN}}$. The lower the FAR value is, the fewer the incorrectly predicted defect-introducing commits that developers need to review. In other words, a low FAR value indicates that developers will spend less effort on reviewing the incorrectly predicted defect-introducing commits.
4) Distance-to-Heaven (d2h) [1] is a root mean square of recall and FAR values, which can be computed as $\sqrt{\frac{(1 - \text{Recall})^2 + (0 - \text{FAR})^2}{2}}$. A d2h value of 0 indicates that an approach can correctly predict all defect-introducing commits without any false positive. A high d2h value indicates that an approach is far from perfect, e.g., achieving a high recall value but also have high FAR value and vice versa.
5) Precision measures the ability of an approach to correctly predict defect-introducing commits, which can be calculated as follows: Precision $= \frac{\text{TP}}{\text{TP} + \text{FP}}$. The higher

Fig. 2: (RQ1) The evaluation result of our JITLine approach compared with the state-of-the-art approaches for Just-In-Time defect prediction (i.e., CC2Vec(Train only), DeepJIT, and EALR).

precision, the better model to correctly predict defect-introducing commits.

6) Recall measures the ability to correctly retrieve defect-introducing commits when making a prediction. The calculation of this measure is $\text{Recall} = \frac{\text{TP}}{\text{TP}+\text{FN}}$. High recall indicates that the model can obtain a lot of defect-introducing commits during prediction.

<u>Results</u>. **Our JITLine approach achieves an AUC 28%-73% higher and an F-measure 26%-38% higher than the state-of-the-art approaches (i.e., CC2Vec).** Figure 2 presents the experimental results of our approach and the state-of-the-art approaches with respect to six evaluation measures, i.e., AUC, FAR, d2h, precision, and recall for Openstack and Qt. We find that our JITLine approach achieves the highest AUC value of 0.83 for Openstack and 0.82 for Qt, which is 1%-8% higher than CC2Vec, 8%-10% higher than DeepJIT, and 28%-73% higher than EALR. We also find that our JITLine approach achieves the highest F-measure value of 0.33 for Openstack and 0.24 for Qt, which are 26%-38% higher than CC2Vec, 300%-3,300% higher than DeepJIT, and 500%-3,200% higher than EALR. This finding indicates that our approach outperforms the state-of-the-art approaches in terms of AUC and F-measure.

**Our JITLine approach achieves a False Alarm Rate (FAR) 94%-97% lower than the CC2Vec approach.** We find that our JITLine approach achieves a False Alarm Rate (FAR) of 0.05 for Openstack and 0.02 for Qt, which is similar to DeepJIT (FAR=0.01) and EALR (FAR=0). Similarly, our JIT-Line approach also achieves a d2h of 0.52 for OpenStack and 0.59 for Qt, which is lower than the state-of-the-art approaches (i.e., DeepJIT and EALR). However, we observe that the d2h of our approach for Qt project is higher than the CC2Vec approach. For Qt project, we find that the lower d2h value of the CC2Vec approach has to do with the high recall value of 0.96—i.e., the CC2Vec approach predicts most of the commits as defect-introducing, but 63%-87% of them are incorrect (i.e., many of them are false positives)— indicating that developers may spend unnecessary effort to inspect actual clean commits

that are incorrectly predicted as defect-introducing commits when the CC2Vec approach was used. On the other hand, the high d2h value of our approach has to do with the low recall of 0.16, but our approach achieves a low FAR of 0.02, indicating that the predictions from our JITLine approach is less likely to predict actual clean commits as defect-introducing. After considering both the ability of identifying defect-introducing commits (i.e., Recall) and the additional costs (i.e., FAR), our approach still outperforms state-of-the-art approaches (i.e., CC2Vec (only for OpenStack), DeepJIT, and EALR).

*(RQ2) Is our JITLine more <u>cost-effective</u> than the state-of-the-art JIT defect prediction <u>approaches</u>?*

<u>Approach</u>. To answer this RQ, we evaluate our JITLine and compare with the four state-of-the-art JIT defect prediction approaches (as mentioned in RQ1) using the following cost-effective measures [1, 11, 17, 24, 44]:

1) PCI@20%LOC measures the proportion of actual defect-introducing commits that can be found given a fixed amount of effort, i.e., the Top 20% LOC of the whole project. A high value of PCI@20%LOC indicates that an approach can rank many actual defect-introducing commits so developers will spend less effort to find actual defect-introducing commits.

2) Effort@20%Recall measures the amount of effort (measured as LOC) that developers have to spend to find the actual 20% defect-introducing commits divided by the total changed LOC of the whole testing dataset. A low value of Effort@20%Recall indicates that the developers will spend a little amount of effort to find the 20% actual defect-introducing commits.

3) $P_{opt}$ is defined as 1-$\Delta_{opt}$, where $\Delta_{opt}$ is the area of the effort-based (i.e., churn) cumulative lift chart between an optimal model and a prediction model. The effort-based (i.e., churn) cumulative lift chart is the relationship between the cumulative percentage of defect-introducing commits from a prediction model ($y$-axis) and the cumulative percentage of the inspection effort ($x$-axis). Similar to prior studies [1, 24, 44], we use the normalized version

(a) OpenStack



(b) Qt

Fig. 3: (RQ2) The cost-effectiveness of our JITLine approach compared to the state-of-the-art approaches for Just-In-Time defect prediction with respect to PCI@20%Recall, Effort@20%Recall, and $P_{Opt}$.

of $P_{opt}$, which is defined as $1 - \frac{\text{Area(Optimal)} - \text{Area(Our)}}{\text{Area(Optimal)} - \text{Area(Worst)}}$. For the *optimal* model and the *worst* model, all commits are ranked by the actual defect density in descending and ascending order, respectively. For *our* model, all commits are ranked by the estimated defect density $\left(\frac{Y(m)}{\#\text{LOC}(c)}\right)$ in descending order.

Results. **Our JITLine approach is 17%-51% more cost-effective than the state-of-the-art approaches in term of PCI@20%LOC.** Figure 3 presents the cost-effectiveness of our JITLine approach compared to the state-of-the-art approaches for Just-In-Time defect prediction with respect to the PCI@20%LOC, Effort@20%Recall and $P_{opt}$ measures. We find that our JITLine approach is more cost-effective than the state-of-the-art approaches for three cost-effectiveness measures. We find that our JITLine achieves a PCI@20%LOC of 0.56 for Openstack and 0.70 for Qt, while the state-of-the-art achieves a PCI@20%LOC of 0.06-0.37 for OpenStack and 0.06-0.60 for Qt. This finding indicates that given a fixed amount of inspection effort at 20%LOC, our JITLine approach can correctly predict 17%-51% higher number of actual defect-introducing commits than the state-of-the-art approaches.

**Our JITLine approach can save the amount of effort by 89%-96% to find the same number of actual defect-introducing commits (i.e., 20% Recall) when compared to the state-of-the-art approaches.** Our JITLine approach achieves an Effort@20%Recall of 0.04 for Openstack and and 0.02 Qt, while the state-of-the-art approaches achieve an Effort@20%Recall of 0.11-0.36 for Openstack, and 0.03-0.53

TABLE III: (RQ3) The average CPU and GPU computational time (minutes±95% Confidence Interval) of the model training of JIT defect prediction approaches after repeating the experiment 5 times.

| | CPU | | GPU | |
|---|---|---|---|---|
| | Openstack | Qt | Openstack | Qt |
| **JITLine** | 36±1 secs | 175±1 secs | - | - |
| **DeepJIT** | 70±7 mins | 143±7 mins | 2±0.01 mins | 5±0.01 mins |
| **CC2Vec** | 146±16 mins | 300±6 mins | 13±0.05 mins | 30±0.10 mins |
| **EARL** | 8±1 secs | 97±1 secs | - | - |

for Qt. Similarly, our JITLine approach achieves a $P_{opt}$ of 0.82 for OpenStack and 0.89 for Qt, which is 116% and 178% higher than the state-of-the-art approaches for OpenStack and Qt, respectively. In particular, our $P_{opt}$ is 7% to 19% higher than EALR, 116% to 178% higher than DeepJIT, and 105% to 112% higher than CC2Vec. This finding suggests that, to find the same amount of actual defect-introducing commits, our JITLine approach can reduced the amount of effort by 85% and 96% when compared to the state-of-the-art approaches, which may provide the best return on investment.

### (RQ3) Is our JITLine faster than the state-of-the-art JIT defect prediction approaches?

Approach. To answer this RQ, we measure the CPU computational time of the model training of our approach, and the CPU and GPU computational time of the model training of deep learning approaches (i.e., DeepJIT and CC2Vec). For our approach, we set `n_jobs` argument of the `RandomForestClassifier` function of Scikit-Learn library to -1 to ensure that all available CPU cores are used in parallel. For the deep learning baselines, we use `cpu` function provided by the Pytorch deep learning library to ensure that all available CPU cores are used in parallel. We perform the experiment using the following equipment: AMD Ryzen 9 5950X 16 Cores/32 Threads Processor, RAM 64GB, NVIDIA GeForce RTX 2080 Ti 11GB. To ensure that our measurement is accurate and strictly controlled, we reserve the computing resources and ensure that the resources are idle with no other running tasks. To combat the randomization bias, we repeat the experiment 5 times.

Results. **Our JITLine approach is 70-100 times faster than the deep learning approaches for Just-In-Time defect prediction.** Table III presents the average CPU and GPU computational time (minutes) of the model training of JIT defect prediction approaches after repeating the experiment 5 times. We find that the model training time of our JITLine approach takes approximately 1-3 minutes, while the model training time of the deep learning approaches for Just-In-Time defect prediction require 1-5 hours (70 to 300 minutes). Given the same running cost (on CPU), this finding suggests that our approach is more cost-efficient than the deep learning approaches.

The computation time of the deep learning approaches can be accelerated by using a high-end GPU hardware. However, we find that the model training time of the deep learning approaches on the GPU device is relatively faster than using

Fig. 4: (RQ4) The results of our JITLine at the line level when compared to the N-gram-based line-level JIT defect prediction approach of Yan *et al.* [43] with respect to Top-10 Accuracy($\nearrow$), Recall@20%Effort($\nearrow$), Effort@20%Recall($\searrow$), and IFA($\searrow$). The higher ($\nearrow$) or the lower ($\searrow$) the values are, the better the approach is.

the CPU hardware with an additional GPU cost. Nevertheless, the model training time of deep learning approaches on GPU (2-30 minutes) still takes relatively longer than the model training time of our approach on CPU (1-3 minutes).

### (RQ4) How effective is our JITLine for prioritizing defective <u>lines</u> of a given defect-introducing commit?

Approach. To address this RQ, we first need to collect the line-level ground-truth data. To do so, we start from cloning a git repository of the studied projects. Then, we use Py-Driller [34], a Python library for mining GitHub repository, to identify defect-fixing commits that are associated with each defect-introducing commit that is provided by McIntosh and Kamei [23]. Once identified, we examine the diff (a.k.a. code changes) made by the defect-fixing commits to identify lines that are modified/deleted by defect-fixing commits. Similar to prior work [4, 31], the lines that were modified or deleted by defect-fixing commits are identified as defective lines, otherwise clean. Then, we compare our JITLine with the state-of-the-art line-level JIT defect prediction approach by Yan *et al.* [43]. We implement the N-gram approach using the implementation provided by Hellendoorn *et al.* [8], Since Yan *et al.* [43] found that the Jelinek-Mercer (JM) smoothing method is the best choice, and the N-gram length has no substantial impact on the average performance, we followed their advice by using the Jelinek-Mercer (JM) smoothing method and the N-gram length of 6. Finally, we evaluate our approach and Yan *et al.* [43] using the following evaluation measures at the line level [42, 43]:

1) Top-10 Accuracy measures the proportion of actual defective lines that are ranked in the top-10 ranking. Traditionally, developers may need to inspect all changed lines for a given commit—which is not ideal when SQA

resources are limited. A high top-10 accuracy indicates that many of the defective lines are ranked at the top, which is considered effective.

2) Recall@20%LOC measures the proportion of defective lines that can be found (i.e., correctly predicted) given a fixed amount of effort (i.e., the top 20% of changed lines of a given defect-introducing commit). A high value of Recall@20%LOC indicates that an approach can rank many actual defective lines at the top.

3) Effort@20%Recall$_{line}$ measures the percentage of the amount of effort that developers have to spend to find the actual 20% defective lines of a given defect-introducing commit. A low value of Effort@20%Recall$_{line}$ indicates that the developers will spend a little amount of effort to find the 20% actual defective lines.

4) Initial False Alarm (IFA) measures the number of clean lines that developers need to inspect until finding the first actual defective line for a given commit. A low IFA value indicates that developers only spend time inspecting only a few number of clean lines to find the first actual defective line.

Results. **Our JITLine approach is 133%-150% more accurate than the baseline approach by Yan *et al.* [43] for identifying actual defective lines in the top-10 recommendations.** Figure 4 shows that our approach achieves a median Top-10 Accuracy of 0.7 for OpenStack and 0.5 for Qt, while the baseline approach achieves a Top-10 Accuracy of 0.3 for Openstack and 0.2 for Qt. In addition, we find that our JITLine approach can find actual defective lines 25%-50% higher than the baseline approach, given the same amount of effort at 20%LOC. Figure 4 shows that our approach achieves a median Recall@20%LOC of 0.20 for OpenStack and 0.21 for Qt, while the baseline approach achieves a median Recall@20%LOC of 0.16 for OpenStack and 0.14 for Qt.

Our Wilcoxon signed-ranked test also confirms that the difference of Top-10 Accuracy and Recall@20%Effort between our approach and the baseline is statistically significantly ($p$-value $< 0.05$) with a Cliff's $|\delta|$ effect size of large ($|\delta| = 0.49 - 0.67$) for both Top-10 Accuracy and Recall@20%LOC.

**Our JITLine approach requires 17%-27% less amount of effort than the baseline approach in order to find the same amount of actual defective lines.** Figure 4 shows that our approach achieves a median Effort@20%Recall$_{line}$ of 0.20 for Openstack and 0.19 for Qt, while the baseline approach achieves a median Effort@20%Recall$_{line}$ of 0.24 for OpenStack and 0.26 for Qt. Similarly, our approach achieves a median IFA of 0 for OpenStack and 1 for Qt, while the baseline approach achieves a median IFA of 3 for OpenStack and 4 for Qt. Our Wilcoxon signed-ranked test also confirms that the difference of Effort@20%Recall$_{line}$ and IFA between our approach and the baseline is statistically significant ($p$-value $< 0.05$) with a Cliff's $|\delta|$ effect size of large ($|\delta| = 0.52 - 0.69$) for Effort@20%Recall$_{line}$ and a Cliff's $|\delta|$ effect size of medium ($|\delta| = 0.36 - 0.39$) for IFA.

## VII. DISCUSSION

### A. Implications to Practitioners

*Our JITLine approach may help practitioners to better prioritize defect-introducing commits and better identify defective lines,* since we find that our JITLine approach outperforms (RQ1), more cost-effective (RQ2), faster (RQ3), and more fine-grained (RQ4) than the state-of-the-art approaches (i.e., EALR, CC2Vec, and DeepJIT). Traditionally, Just-In-Time defect prediction methods only prioritize defect-introducing commits, saving a lot of code inspection effort. However, we find that the average ratio of actual defective lines for each commit is 50%. Thus, developers still spend unnecessarily effort on inspecting clean lines. In addition to predict defect-introducing commits, our JITLine approach can also accurately predict defective lines within a defect-introducing commit, saving 17%-20% effort that developers need to spend when compared to the baseline approach [43].

### B. Implications to Researchers

*Researchers should consider the key principles of Just-In-Time defect prediction models (i.e., to generate predictions as soon as possible),* since the results of our replication study show that, when excluding testing datasets, the F-measure of CC2Vec approach is decreased by 38.5% for OpenStack and 45.7% for Qt. In reality, it is unlikely that the unlabelled testing dataset would be available beforehand when training JIT models. Thus, when conducting an experiment, testing data should be excluded when developing AI/ML models.

*Researchers should explore simple solutions (i.e., Explainable AI approaches [13, 14, 28, 38, 42]) first over complex and compute-intensive deep learning approaches for SE tasks*, since we find that our JITLine approach outperforms the deep learning approaches for Just-In-Time defect prediction. This recommendation has been advocated by prior studies in other SE tasks [5, 8, 22, 25]. For example, Menzies *et al.* [25] suggested that researchers should explore simple and fast approaches before applying deep learning approaches on SE tasks. Hellendoorn [8] found that a careful implementation of NLP approaches outperform deep learning approaches. Liu *et al.* [22] found that simple $k$-nearest neighbours approach outperforms neural machine translation approaches.

### C. Threats to Validity

*Threats to construct validity* relates to the impact of parameter settings of the techniques that our approach relies upon (i.e, SMOTE, DE, Random Forest, and LIME) [6, 39, 40]. To mitigate this threat, we apply a Differential Evolution algorithm to optimize the parameter setting of the SMOTE technique. We use the parameter settings of DE, suggested by Agrawal *et al.* [2]. We use the default settings of LIME (i.e., the number of samples = 5,000). For the baseline approaches, we use the best parameter settings provided by the implementation of the DeepJIT [9] and CC2Vec approaches [10].

Prior work raised concerns that the ground-truths data collection of defect-introducing commits could be delayed [3, 36]. Thus, it is possible that our studied JIT datasets may be missing some of the false negative commits when defects are not fixed (i.e., defect-fixing commits that are not yet fixed). However, the goal of this paper is not to improve the data construction approach. Instead, we use the same datasets that were used in the prior work for a fair comparison. Thus, future work should consider addressing this concern.

*Threats to external validity* relates to the limited number of the studied datasets (i.e., OpenStack and Qt) to ensure a fair comparison with the CC2Vec approach [10]. Thus, other commit-level datasets can be explored in future work.

*Threats to internal validity* relates to the randomization of several techniques that our approach relies upon [21]. After we repeat our experiments with different random seeds, we observe minor differences (e.g., ±0.01 for AUC). Nevertheless, our JITLine approach is still the best performer for all RQs. The used random seed number is reported in our replication package at Zenodo: http://doi.org/10.5281/zenodo.4433498.

We follow the experimental setting of the original study [9, 10] (i.e., one single training/testing data split without cross-validation). Therefore, statistical analysis and effect size analysis are not applied for RQ1, RQ2, and RQ3, since we have only one performance value for each project.

## VIII. CONCLUSIONS

In this paper, we propose JITLine approach, a machine learning-based JIT defect approach for predicting defect-introducing commits and identifying defective lines that are associated with that commit. Then, we conduct our empirical study to demonstrate that our JITLine approach is better (RQ1), more cost-effective (RQ2), faster (RQ3) and more fine-grained (RQ4) than the state-of-the-art JIT defect prediction approaches (i.e., EARL, DeepJIT, and CC2Vec).

Therefore, our JITLine approach may help practitioners to better prioritize defect-introducing commits and better identify defective lines. In addition, our results highlight the negative impact of excluding testing datasets in model training and the importance of exploring simple solutions (e.g., explainable AI approaches) first over complex and compute-intensive deep learning approaches.

## REFERENCES

[1] A. Agrawal, W. Fu, D. Chen, X. Shen, and T. Menzies, "How to"dodge" complex software analytics," *Transactions on Software Engineering (TSE)*, 2019.

[2] A. Agrawal and T. Menzies, "Is "better data" better than "better data miners"?" in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2018, pp. 1050–1061.

[3] G. G. Cabral, L. L. Minku, E. Shihab, and S. Mujahid, "Class imbalance evolution and verification latency in just-in-time software defect prediction," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2019, pp. 666–676.

[4] D. A. Da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *Transactions on Software Engineering (TSE)*, pp. 641–657, 2016.

[5] W. Fu and T. Menzies, "Easy over hard: A case study on deep learning," in *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*, 2017, pp. 49–60.

[6] W. Fu, T. Menzies, and X. Shen, "Tuning for software analytics: Is it really necessary?" *Information and Software Technology (IST)*, pp. 135–146, 2016.

[7] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi, "An empirical study of just-in-time defect prediction using cross-project models," in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2014, pp. 172–181.

[8] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*, 2017, pp. 763–773.

[9] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2019, pp. 34–45.

[10] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "CC2Vec: Distributed representations of code changes," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2020, pp. 518–529.

[11] Q. Huang, X. Xia, and D. Lo, "Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 159–170.

[12] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2017, pp. 135–146.

[13] J. Jiarpakdee, C. Tantithamthavorn, and J. Grundy, "Practitioners' Perceptions of the Goals and Visual Explanations of Defect Prediction Models," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2021, p. To Appear.

[14] J. Jiarpakdee, C. Tantithamthavorn, H. Khanh Dam, and J. Grundy, "An empirical study of model-agnostics techniques for defect prediction models," *IEEE Transactions on Software Engineering (TSE)*, 2020.

[15] M. Jimenez, R. Rwemalika, M. Papadakis, F. Sarro, Y. Le Traon, and M. Harman, "The importance of accounting for real-world labelling when predicting software vulnerabilities," in *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 695–705.

[16] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2010, pp. 1–10.

[17] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *Transactions on Software Engineering (TSE)*, pp. 757–773, 2012.

[18] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: clean or buggy?" *Transactions on Software Engineering (TSE)*, pp. 181–196, 2008.

[19] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 489–498.

[20] G. Lemaître, F. Nogueira, and C. K. Aridas, "Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning," *Journal of Machine Learning Research*, pp. 1–5, 2017.

[21] C. Liem and A. Panichella, "Run, forest, run? on randomization and reproducibility in predictive software engineering," *arXiv preprint arXiv:2012.08387*, 2020.

[22] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: how far are we?" in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2018, pp. 373–384.

[23] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," *Transactions on Software Engineering (TSE)*, pp. 412–428, 2017.

[24] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2010, pp. 107–116.

[25] T. Menzies, S. Majumder, N. Balaji, K. Brey, and W. Fu, "500+ times faster than deep learning:a case study exploring faster methods for text mining stackoverflow," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2018, pp. 554–563.

[26] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *Journal of Systems and Software*, pp. 22 – 36, 2019.

[27] M. Rahman, D. Palani, and P. C. Rigby, "Natural software revisited," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2019, pp. 37–48.

[28] D. Rajapaksha, C. Tantithamthavorn, J. Jiarpakdee, C. Bergmeir, J. Grundy, and W. Buntine, "SQAPlanner: Generating Data-Informed Software Quality Improvement Plans," 2021.

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

[30] M. T. Ribeiro, S. Singh, and C. Guestrin, "Why should I trust you?" explaining the predictions of any classifier," in *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2016, pp. 1135–1144.

[31] G. Rodríguez-Pérez, G. Robles, and J. M. González-Barahona, "Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm," *Information and Software Technology (IST)*, pp. 164–176, 2018.

[32] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim, "Reducing features to improve code change-based bug prediction," *Transactions on Software Engineering (TSE)*, pp. 552–569, 2012.

[33] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, 2005, p. 1–5.

[34] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018, pp. 908–911.

[35] R. Storn and K. Price, "Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces," *Journal of Global Optimization*, pp. 341–359.

[36] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *Proceedings of the International Conference on Software Engineering (ICSE)*, vol. 2, 2015, pp. 99–108.

[37] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, "The impact of class rebalancing techniques on the performance and interpretation of defect prediction models," *Transactions on Software Engineering (TSE)*, 2020.

[38] C. Tantithamthavorn, J. Jiarpakdee, and J. Grundy, "Explainable AI for Software Engineering," *arXiv preprint arXiv:2012.01614*, 2020.

[39] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction models," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2016, pp. 321–332.

[40] ——, "The impact of automated parameter optimization on defect prediction models," *Transactions on Software Engineering (TSE)*, pp. 683–711, 2018.

[41] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright *et al.*, "Scipy 1.0: fundamental algorithms for scientific computing in python," *Nature methods*, pp. 261–272, 2020.

[42] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto, "Predicting defective lines using a model-agnostic technique," *Transactions on Software Engineering (TSE)*, 2020.

[43] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, and S. Li, "Just-in-time defect identification and localization: A two-phase framework," *Transactions on Software Engineering (TSE)*, 2020.

[44] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models," in *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2016, pp. 157–168.