IEICE TRANSACTIONS

on Information and Systems

VOL. E106-D NO. 2
FEBRUARY 2023

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY

The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER Special Section on Empirical Software Engineering

Commit-Based Class-Level Defect Prediction for Python Projects

Khine Yin MON[†a)], **Masanari KONDO**[††], *Nonmembers*, **Eunjong CHOI**[†††], *and* **Osamu MIZUNO**[†††], *Members*

SUMMARY Defect prediction approaches have been greatly contributing to software quality assurance activities such as code review or unit testing. Just-in-time defect prediction approaches are developed to predict whether a commit is a defect-inducing commit or not. Prior research has shown that commit-level prediction is not enough in terms of effort, and a defective commit may contain both defective and non-defective files. As the defect prediction community is promoting fine-grained granularity prediction approaches, we propose our novel class-level prediction, which is finer-grained than the file-level prediction, based on the files of the commits in this research. We designed our model for Python projects and tested it with ten open-source Python projects. We performed our experiment with two settings: setting with product metrics only and setting with product metrics plus commit information. Our investigation was conducted with three different classifiers and two validation strategies. We found that our model developed by random forest classifier performs the best, and commit information contributes significantly to the product metrics in 10-fold cross-validation. We also created a commit-based file-level prediction for the Python files which do not have the classes. The file-level model also showed a similar condition as the class-level model. However, the results showed a massive deviation in time-series validation for both levels and the challenge of predicting Python classes and files in a realistic scenario.
key words: *defect prediction; fine-grained prediction, empirical software engineering, mining software repositories*

1. Introduction

Software developers continually modify the source code to fix the existing software defects and add new features. However, these modifications usually lead to the introduction of new defects, which can decrease the quality of the software [1]. Software quality assurance activities (SQA) are necessary to guarantee the achievement of premium software products. Nevertheless, these kinds of activities are challenging due to the balance between limited resources and time-to-market requirements [2]. Defect prediction technology arises to assist SQA in predicting the software's risky parts. Therefore, the practitioners can allocate their quality assurance efforts more effectively, e.g., testing and code reviews [3].

In 2019, Pascarella et al. claimed that the commit-level defect prediction is coarse because a commit can contain multiple files, and all the files within a defective commit might not be defect-prone [4]. Therefore, they investigated for the proportion of the actual defective files in a defective commit and reported that almost 43% of the changed files within a defective commit are defective. Further, they mentioned that 42% of defective commits were composed of a mixture of both defective and non-defective files in their studied subjects. Therefore, they proposed a two-phase fine-grained just-in-time prediction model, which identifies the defect-prone files within a defective commit. However, in a real-world scenario, the file-level is still coarse. A file can have multiple classes, such as the Math project in the Defects4j dataset, and the developers have to take a considerable amount of time to inspect all the codes in the entire file [5]. The finer-grained level such as class-level and function-level than the file-level should be oriented for this approach. Hence, our ultimate goal is to develop a finer-grained two-phase defect prediction, which uses the commit information.

To this aim, we first examined which granularity, i.e., which level of defect prediction, was appropriate for our approach. We surveyed the popularity among fine-grained models, and our survey result led us to choose the class-level granularity to build our intended model. Subsequently, we proposed our novel commit-based class-level defect prediction approach and experimented with two settings: product metrics only approach and product metrics plus commit information approach. We experimented with our study with three classifiers, random forest [6], logistic regression [7], and support-vector machine [8], and validated with two validation strategies, 10-fold cross-validation [9] and time-series validation [10]. We also developed a commit-based file-level approach for the Python files that do not contain classes. Finally, we compared the results. The comparison results described that random forest outperforms all the classifiers, and the class-level defect prediction approach can be improved by adding commit information when we validate with 10-fold cross-validation. However, this finding has a significant difference when we validate with a real-time scenario and shows space for contributing more to the area of predicting Python classes and files with time-series data.

The three main contributions of this paper are as follows:

- We have proposed the commit-based class-level and

Manuscript received March 31, 2022.
Manuscript revised August 21, 2022.
Manuscript publicized November 14, 2022.
[†]The author is with the Graduate School of Science and Technology, Kyoto Institute of Technology, Kyoto-shi, 606–8585 Japan.
[††]The author is with the Faculty of Information Science and Electrical Engineering, Kyushu University, Fukuoka-shi 819-0395 Japan.
[†††]The authors are with the Faculty of Information and Human Sciences, Kyoto Institute of Technology, Kyoto-shi, 606–8585 Japan.
a) E-mail: k-yinmon@se.is.kit.ac.jp
DOI: 10.1587/transinf.2022MPP0003

file-level defect prediction model (i.e., the classes and the files are extracted from the commits) and tested the model with the two experimental settings (i.e., testing with only product metrics setting and both the commit information and the product metrics)

- We have evaluated and compared these models with different classifiers and different validation strategies and reported the result.

- We have performed the systematic literature review among the granularity of the fine-grained defect prediction models.

Structure of the paper: Sect. 2 reports the background of this study. Section 3 is the methodology section which includes an explanation of the research questions, the literature review of fine-grained prediction models, the process of our approach, and the metrics used in this study. Section 4 describes the detailed information of studied subject systems and an overview of the experiment. Section 5 is the result section. Section 6 explains the threats that might influence our findings. Section 7 concludes the paper.

2. Background

2.1 Defect Prediction

Defect prediction approaches can be divided into two categories: long-term prediction approaches and short-term prediction approaches [4]. Long-term prediction approaches analyze the information of previous releases and predict the defectiveness of future releases. One of the significant limitations of the long-term prediction approach is that predictions are made very late in the software development cycle. Meanwhile, the short-term prediction approaches predict whenever the code is changed and saved, such as session time [11] or commit time, as well as provide immediate feedback for the defect [4].

2.2 Just-in-Time Defect Prediction

Just-in-time defect prediction models are included in the short-term prediction category and assist in making a prediction about the defectiveness of a commit. However, one of the drawbacks of the just-in-time prediction models is that when any code in a commit relates to the defects, the whole commit is treated as the defective commit. Consequently, the developer has to inspect all codes within a commit. Recently, the researchers addressed this situation by proposing models which can give further investigation about the potential defectiveness of the codes with a commit [4], [12], [13].

2.3 Defect Prediction Size Granularity

Widespread studies of defect prediction approaches are proposed based on machine learning techniques. The mainstream process of machine learning-based defect prediction approaches generally includes generating instances from software archives, labeling these instances, preprocessing (optional), model training, and finally predicting the new instances by the trained model [14]. An instance can be a subsystem, a component, a file, a class, a function, a line, or a token, as Fig. 1. The coarsest granularity can be denoted as subsystem level, while the finest is the token. Recently, the fine-grained granularity approaches, such as class-level, method-level, and line-level, have been promoted because some studies proved that the fine-grained defect prediction approaches are more cost-effective than the coarse-grained ones [5], [15].

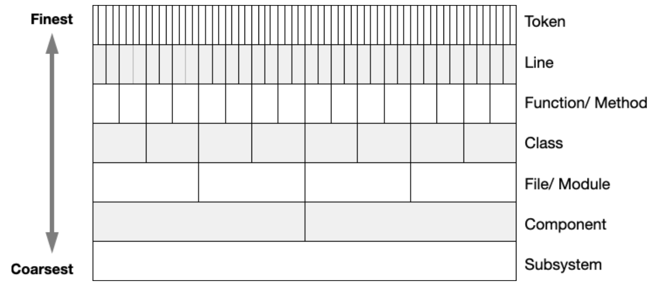

Fig. 1 The granularity of defect prediction models.

3. Methodology

This section presents our research questions, literature review about fine-grained defect prediction models, outlines of our method, independent variables, and studied classifiers.

3.1 Research Questions

This research aims to build a two-phase fine-grained defect prediction model that uses the commit information. The prior study proposed a concept of commit-based file-level defect prediction, which identifies the defect-prone files within a commit [4]. Nevertheless, the effort to inspect all the non-defective and defective code elements within a file should be considered. However, it is unclear which granularity is popular and preferable in the within-project defect prediction research community for our proposed two-phase fine-grained prediction approach. For that reason, we decided to survey the popular granularity of fine-grained defect prediction models, and we set our first question as follows. In this RQ, we summarize the size of the granularity of the prediction target in prior studies.

- **RQ 1: Which granularity level of fine-grained defect prediction does the research community of the defect prediction orient the most?**

Based on the answer to research question 1, we built our prediction model. Finally, we measured our model's performance and represented the results as the answer to research question 2.

- **RQ 2: How well can our proposed two-phase model predict for a class of a commit?**

Our study aims to exploit the commit information for the fine-grained defect prediction model. According to the result of RQ1, we chose the class-level as the target fine-grained granularity of our study. However, our target projects are Python and there are Python files that do not have classes. Therefore, we added a commit-based file-level defect prediction and compared it with our commit-based class-level model for defect prediction in Python.

- **RQ 3: How is the performance of commit-based class-level defect prediction when comparing to that of commit-based file-level defect prediction?**

3.2 Literature Review about the Most Popular Fine-Grained Defect Prediction Models for within Project Area

To gain insight knowledge about defect prediction granularity and find out the most popular granularity in the defect prediction community, we performed a literature review. Since this study was not intended to become a systematic literature review for a wide area of defect prediction research field, we defined the area scope of our literature review. In Pascarella et al.'s approach [4], they set file-level as the fine-grained granularity, and we aimed to improve their approach by developing a finer-grained prediction model than the one of their model. Therefore, we counted for class, function, method, line, and token levels, that are finer than the file-level. We searched the papers in the following six venues, the premier publication venues in the software engineering research community, from 2015 to 2020. Among these venues, the two venues are for journals, and the rest are for conferences.

- ASE - IEEE/ACM International Conference on Automated Software Engineering
- ESEC/FSE - ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering
- ICSE - ACM/IEEE International Conference on Software Engineering
- SANER - IEEE International Conference on Software Analysis, Evolution and Reengineering
- TOSEM - ACM Transactions on Software Engineering and Methodology
- TSE - IEEE Transactions on Software Engineering

We used the keywords "defect," "bug," "fault," and "prediction" to search in IEEE/ACM digital libraries. We filtered the relevant papers by reading the titles, abstract, and keywords and collected the titles of all resulting papers. Since our goal is to build a prediction model for within-project setting by the machine learning technique, we excluded some papers such as papers which are using cross-project settings [16]–[19], and deep learning techniques. Moreover, we skipped the papers that are not available for the full text. Finally, we downloaded the full text of the rest papers and found out the granularity of the predicted part.

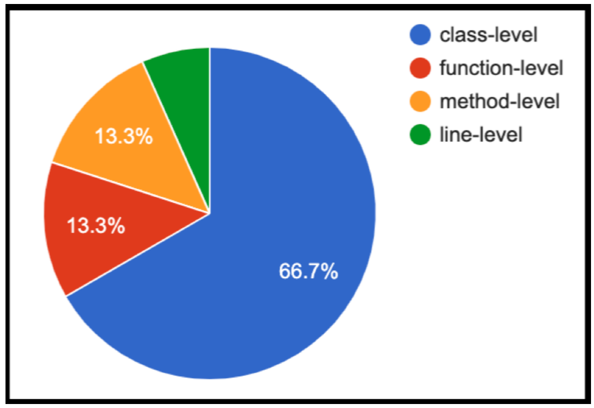

Fig. 2 66.7% belong to class, 13.3% belong to function, 13.3% belong to method, and 6.7% belong to line levels.

Figure 2 shows the result of our literature review. We finally found 15 papers. Among them, 66.7% (10 papers) are class-level, 13.3% (2 papers) are function-level, 13.3% (2 papers) are method-level, and 6.7% (1 paper) is line-level. As we discovered the most contributed granularity of defect prediction models, we recognized class-level granularity as the answer to our first research question. Thus, we built our proposed model for class-level defect prediction granularity.

In addition, we also discovered that the experimented systems of the observed ten class-level defect prediction papers are mainly Java projects [20]–[29]. Despite the increasing popularity of the Python programming language in various domains such as machine learning and deep learning [30], the defect prediction research community has not paid attention to Python projects, to our knowledge. To remedy this, we applied Python projects in our study.

> RQ 1: The class level is the most popular granularity among the fine-grained prediction models from 2015 to 2020.

3.3 Commit-Based Class-Level and File-Level Defect Prediction

Our commit-based class/file-level defect prediction model consists of two phases: (1) the commit-level defect prediction phase and (2) the class/file-level defect prediction phase. We identified defective commits in the first phase and identified defective classes/files on the defective commits in the second phase. We describe the steps to build our model (Fig. 3) as follows.

1. We collected all the defective and non-defective commits from a target project.
2. All the classes of the modified files of the commits were extracted.
3. Product metrics of classes and files, and commit information (e.g., number of added lines) were calculated as

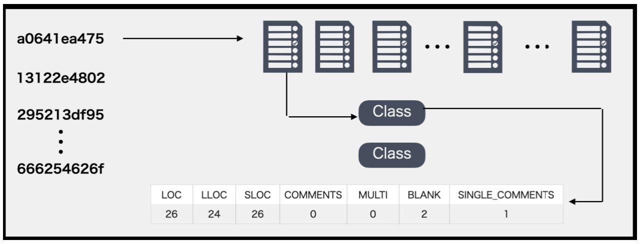

Fig. 3 The process flow of our commit-based class-level defect prediction approach.

the studied metrics.

4. We made a dataset that contains both commit information and product metrics.
5. Our model was trained and tested with the gained dataset.

Rosen et al. proposed a tool, Commit Guru, to automatically identify and predict the defect-prone commits on projects [31]. Our first step used Commit Guru [31] to extract defective and non-defective commits for such projects.

Our second step started with extracting the modified files of the projects. We cloned GitHub repositories for each project. We applied PyDriller [32] to each commit of the cloned repository to get modified files. We parsed all classes in the modified files using an AST tool of Python [33].

In the third step, we calculated product metrics for each modified file and class by Radon [34]. Also, we got the metrics of commit information from each commit with Commit Guru. Product metrics and the commit information are described in Sect. 3.4.

In the fourth step, a new dataset was acquired by concatenating the calculated product metrics with the commit information for modified files and classes, respectively.

Finally, we trained and tested classifiers (described in Sect. 3.5) based on the dataset. For the commit-based class-level defect prediction, we used the dataset in which product metrics were computed from modified classes; for the commit-based file-level defect prediction, we used the dataset in which product metrics were computed from modified files. For both cases, we built the defect prediction model to identify defective classes/files based on the commit information.

3.4 Independent Variables

In this paper, we prepared two sets of independent variables: product metrics and commit information. The independent variables are for extracting and measuring the characteristics of the classes/ files in a commit. In this study, we considered the most basic product metrics which can show the minimum performance of our model. For the commit information, we adopted the widely-used 14 change-level metrics proposed by Kamei et al. [3]. The metrics included in Table 1 and Table 2 show the overview. The details are as follows.

Product metrics: LOC [35], Halstead [36], and McCabe's Cyclomatic Complexity [37] are included in this set. These metrics are extracted with the assistance of the Python tool, Radon†. Table 1 reports the information of these metrics.

Commit information: Just-in-time defect prediction models usually use the commit information; therefore, we used the following listed just-in-time defect prediction metrics [3] as the commit information: NS, ND, NF, Entropy, LT, LA, LD, FIX, NDEV, AGE, NUC, EXP, REXP, and SEXP. The metrics and their description are listed in Table 2.

Table 1 List of the product metrics.

Acronym	Name
loc	The number of lines of code (total)
lloc	The number of logical lines of code
sloc	The number of source lines of code (not necessarily corresponding to the LLOC)
comments	The number of Python comment lines
multi	The number of lines which represent multiline strings
single_comments	The number of lines which are just comments with no code
blank	The number of blank lines (or whitespace-only ones)
h1	the number of distinct operators
h2	the number of distinct operands
N1	the total number of operators
N2	the total number of operands
h	the vocabulary, i.e. h1 + h2
N	the length, i.e. N1 + N2
calculated_length	h1 * log2(h1) + h2 * log2(h2)
volume	V = N * log2(h)
difficulty	D = h1 / 2 * N2 / h2
effort	E = D * V
time	T = E / 18 seconds
bugs	B = V / 3000 - an estimate of the errors in the implementation
real_complexity	Cyclomatic Complexity value of a piece of code

Table 2 List of the commit information variables.

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

†https://radon.readthedocs.io/

Table 3 Characteristics of the subject software systems.

Systems	# of Commits	% of Defect-prone Commits	# of Classes	# of Python Files	# of Classes Per Python File
ADSM	3493	24%	229	169	1.36
Axelrod	5539	24%	737	185	3.98
Bitmask_client	3055	18%	188	162	1.16
Galicaster	1786	20%	100	147	0.68
Lisa	3876	18%	1251	502	2.49
Parsl	3724	32%	159	398	0.40
PyBitmessage	2595	30%	473	376	1.26
PythonRobotics	1700	20%	98	214	0.46
TADbit	2503	42%	22	100	0.22
Toil	5649	36%	328	261	1.26

3.5 Studied Classifiers

To build the defect prediction model, we used the following three classifiers: random forest (RF) [6], logistic regression (LR) [7], and support-vector machines (SVM) [8]. These classifiers are applied and expressed as the popular machine learning models for defect prediction in the prior studies [3], [4], [23], [38]. We exploited the Weka toolkit [39] to use these classifiers.

4. Experimental Setup

4.1 Subject Systems

Overall 480 analyzed repositories were available on Commit Guru on July 1, 2022, which were also available on GitHub. We only chose 56 Python projects from these selected, contributing to more than 70% of Python or jupyter notebook codes. From these projects, we filtered out the projects that did not meet the following criteria: the projects 1. should have over 1000 commits, and 2. should include over 10% of defect-prone commits. After filtering with these two criteria, we ended up with 33 projects. Finally, we randomly selected ten projects, the same as the existing work [4]. The selected projects are listed in Table 3. We selected the Python projects for this study because of the need for more research contributing to the area of Python programming language [40]–[42]. In Table 3, we mention the list of the systems with the number of commits, the percentage of defect-prone commits, the number of classes, the number of Python files, and the average number of classes per Python file. In this study, we particularly define .py or .ipynb as the Python files.

4.2 Overview of Experiment

We trained our defect prediction models with two different settings based on the two independent metrics sets: 34-attribute setting (commit information + product metrics) and 20-attribute setting (product metrics). Prior studies do not use the commit information to identify defective classes. Hence, we studied these settings to clarify the impact of the commit information on identifying defective classes.

To validate the defect prediction model, we used two validation strategies: the 10-fold cross-validation [9] and the time-series validation [10].

The 10-fold cross-validation was applied as the validation strategy, as in the prior studies [3], [13]. The 10-fold cross-validation randomly divides the original dataset into ten equal-sized folds. Of the ten folds, one fold is used as the validation data for testing the model, and the remaining nine folds are for training data. The process is repeated ten times, with each fold is applied exactly one time as the validation data. Afterward, the accuracy result is taken as the mean value of the ten times validation.

We also experimented with our models with a time-series validation strategy. We sorted the data rows in our datasets by date and time and divided the datasets in a 7:3 ratio. The training data is 70%, and the testing data is the rest 30%. For example, for the TADbit project, the overall project data is available from 2012-10-24 to 2021-11-03. After sorting and dividing the TADbit dataset, the testing data is from 2019-05-29 to the end, and the training data is from the start to 2019-05-28. In addition, we changed the labels of the rows of the training data set referring to the prior study [10]. The commits become defect-inducing commits because of the defect-fixing commits. Suppose the defect-fixing commits of the defect-inducing commits are found after the period of the training data set. In that case, these defect-inducing commits should not be regarded as defect-inducing commits. These commits should be treated as clean ones because their defects are not yet found. Hence, this validation simulates a more realistic scenario than the 10-fold cross-validation.

5. Result

5.1 RQ 2: How Well Can Our Proposed Two-Phase Model Predict for a Class of a Commit?

We evaluated the performance measures of our prediction model, which was trained and tested by the random forest (RF), logistic regression (LR), and support-vector machines (SVM) with 10-fold cross-validation and time-series validation, as described in Sect. 4.2. Table 4 and Table 5 provide F-measures and AUC-ROC results of our models with the product metrics (20-attribute setting) and the commit information and product metrics (34-attribute setting), respectively. According to the performance of all classifiers, the

Table 4 Performance result of the class-level prediction model (20-attribute setting).

Projects	Classifiers	10-Fold Cross Validation		Time-Series Validation	
		F-measure	AUC-ROC	F-measure	AUC-ROC
ADSM	LR	0.560	0.659	0.653	0.492
	RF	0.693	0.766	0.649	0.496
	SVM	0.685	0.638	0.650	0.499
Axelrod	LR	0.454	0.529	0.320	0.523
	RF	0.557	0.567	0.392	0.492
	SVM	0.525	0.530	0.324	0.492
Bitmask_client	LR	0.437	0.526	0.400	0.491
	RF	0.545	0.543	0.438	0.464
	SVM	0.520	0.522	0.533	0.519
Galicaster	LR	0.518	0.541	0.533	0.532
	RF	0.557	0.583	0.591	0.498
	SVM	0.536	0.540	0.388	0.502
Lisa	LR	0.487	0.545	0.558	0.504
	RF	0.641	0.659	0.570	0.426
	SVM	0.616	0.587	0.564	0.503
Parsl	LR	0.459	0.528	0.401	0.506
	RF	0.582	0.604	0.418	0.488
	SVM	0.577	0.557	0.414	0.493
PyBitmessage	LR	0.480	0.531	0.373	0.475
	RF	0.576	0.588	0.435	0.479
	SVM	0.554	0.556	0.383	0.499
PythonRobotics	LR	0.504	0.561	0.525	0.518
	RF	0.553	0.599	0.516	0.483
	SVM	0.543	0.544	0.552	0.504
TADbit	LR	0.611	0.632	0.523	0.479
	RF	0.665	0.651	0.550	0.541
	SVM	0.641	0.565	0.297	0.493
Toil	LR	0.579	0.537	0.429	0.495
	RF	0.644	0.605	0.632	0.495
	SVM	0.627	0.531	0.637	0.506
Mean	**LR**	**0.509**	**0.559**	**0.472**	**0.502**
	RF	**0.601**	**0.616**	**0.519**	**0.486**
	SVM	**0.582**	**0.557**	**0.474**	**0.501**

Table 5 Performance result of the class-level prediction model (34-attribute setting).

Projects	Classifiers	10-Fold Cross Validation		Time-Series Validation	
		F-measures	AUC-ROC	F-measures	AUC-ROC
ADSM	LR	0.708	0.784	0.691	0.584
	RF	0.989	0.999	0.590	0.395
	SVM	0.870	0.824	0.217	0.459
Axelrod	LR	0.686	0.735	0.553	0.624
	RF	0.985	0.999	0.438	0.606
	SVM	0.766	0.745	0.406	0.511
Bitmask_client	LR	0.688	0.759	0.517	0.642
	RF	0.974	0.996	0.502	0.697
	SVM	0.678	0.670	0.546	0.543
Galicaster	LR	0.725	0.788	0.520	0.454
	RF	0.896	0.950	0.521	0.574
	SVM	0.461	0.541	0.488	0.423
Lisa	LR	0.753	0.802	0.695	0.602
	RF	0.987	0.999	0.559	0.632
	SVM	0.720	0.683	0.455	0.509
Parsl	LR	0.710	0.776	0.335	0.525
	RF	0.925	0.979	0.330	0.581
	SVM	0.693	0.672	0.346	0.486
PyBitmessage	LR	0.677	0.758	0.409	0.501
	RF	0.917	0.976	0.478	0.386
	SVM	0.587	0.614	0.531	0.530
PythonRobotics	LR	0.762	0.849	0.553	0.412
	RF	0.885	0.955	0.539	0.505
	SVM	0.554	0.586	0.537	0.498
TADbit	LR	0.809	0.823	0.470	0.475
	RF	0.816	0.908	0.425	0.369
	SVM	0.605	0.538	0.214	0.484
Toil	LR	0.657	0.713	0.393	0.445
	RF	0.973	0.997	0.307	0.405
	SVM	0.772	0.677	0.306	0.477
Mean	**LR**	**0.718**	**0.779**	**0.514**	**0.526**
	RF	**0.935**	**0.976**	**0.469**	**0.515**
	SVM	**0.671**	**0.655**	**0.405**	**0.492**

random forest gave us the highest result for 10-fold cross-validation. Average F-measure and AUC-ROC values were over 0.935 and 0.976 for the 34-attribute setting, while these values were 0.601 and 0.616 for the 20-attribute setting, respectively.

In 10-fold cross-validation, all classifiers increased their performance by adding the commit information (i.e., changing from the 20-attribute setting to the 34-attribute setting). Indeed, as described above, the average AUC-ROC value for RF rose to 0.976 from 0.616. Also, the average AUC-ROC value rose to 0.779 from 0.559 for LR and 0.655 from 0.557 for SVM. We observed a similar tendency in terms of F1-score.

Nevertheless, in the time-series validation, all classifiers show an average AUC-ROC of around 0.5, which is the worst performance in AUC-ROC. Even if using the commit information, the result is almost the same. This experiment shows that our commit-based class-level defect prediction model works well in a 10-fold validation strategy but underperforms in the time-series validation, which is a more realistic scenario.

> The proposed commit-based class-level defect prediction model with RF performs better than the other classifiers in the 10-fold cross-validation. Also, adding the commit information increases the prediction performance. However, the proposed model does not work well in a realistic scenario (i.e., the time-series validation).

5.2 RQ 3: How is the Performance of Commit-Based Class-Level Defect Prediction When Comparing to That of Commit-Based File-Level Defect Prediction?

Tables 6 and 7 show the performance of the commit-based file-level defect prediction model with the 20-attribute setting and the 34-attribute setting, respectively. We achieved the same conclusion as the ones in RQ2. For example, RF achieves the best performance compared to the other classifiers in the 10-fold cross-validation. Importantly, even if we evaluate the commit-based file-level defect prediction model, which is a coarser grain than class-level, the prediction performance on the time-series validation is the worst in terms of AUC-ROC. Hence, this result and the result in RQ2 imply that the commit-based class/file-level defect prediction model needs more future work to identify defective classes/files in Python in a realistic scenario.

> The proposed commit-based file-level defect prediction model shows the same tendency as the commit-based class-level defect prediction model. Especially the model shows the worst performance in the time-series validation. Hence, identifying defective files/-classes in Python is still a challenging task in defect prediction. Future studies are necessary on this challenge.

6. Threats to Validity

6.1 External Validity

We experimented with ten open-source Python projects with different ratios of defect-prone commits, number of overall

Table 6 Performance result of the file-level prediction model (20-attribute setting).

Projects	Classifiers	10-Fold Cross Validation		Time-Series Validation	
		F-measure	AUC-ROC	F-measure	AUC-ROC
ADSM	LR	0.550	0.590	0.623	0.537
	RF	0.602	0.650	0.632	0.537
	SVM	0.571	0.570	0.559	0.476
Axelrod	LR	0.469	0.538	0.263	0.518
	RF	0.550	0.560	0.394	0.502
	SVM	0.550	0.542	0.297	0.506
Bitmask_client	LR	0.525	0.552	0.455	0.501
	RF	0.556	0.560	0.463	0.474
	SVM	0.504	0.511	0.455	0.460
Galicaster	LR	0.551	0.572	0.516	0.533
	RF	0.561	0.594	0.514	0.494
	SVM	0.536	0.540	0.527	0.481
Lisa	LR	0.621	0.657	0.558	0.623
	RF	0.645	0.694	0.394	0.438
	SVM	0.571	0.577	0.387	0.501
Parsl	LR	0.511	0.551	0.327	0.473
	RF	0.592	0.599	0.365	0.520
	SVM	0.582	0.548	0.307	0.501
PyBitmessage	LR	0.502	0.510	0.354	0.470
	RF	0.557	0.570	0.370	0.466
	SVM	0.548	0.546	0.370	0.466
PythonRobotics	LR	0.531	0.550	0.510	0.502
	RF	0.537	0.572	0.545	0.500
	SVM	0.506	0.512	0.510	0.496
TADbit	LR	0.657	0.662	0.467	0.446
	RF	0.600	0.622	0.247	0.357
	SVM	0.591	0.516	0.377	0.514
Toil	LR	0.568	0.566	0.613	0.510
	RF	0.672	0.637	0.198	0.488
	SVM	0.653	0.566	0.126	0.500
Mean	**LR**	**0.548**	**0.575**	**0.469**	**0.511**
	RF	**0.587**	**0.606**	**0.412**	**0.478**
	SVM	**0.561**	**0.543**	**0.391**	**0.490**

Table 7 Performance result of the file-level prediction model (34-attribute setting).

Projects	Classifiers	10-Fold Cross Validation		Time-Series Validation	
		F-measure	AUC-ROC	F-measure	AUC-ROC
ADSM	LR	0.685	0.750	0.626	0.537
	RF	0.799	0.884	0.638	0.563
	SVM	0.454	0.546	0.493	0.571
Axelrod	LR	0.683	0.734	0.368	0.393
	RF	0.889	0.955	0.592	0.697
	SVM	0.457	0.523	0.258	0.500
Bitmask_client	LR	0.671	0.719	0.536	0.598
	RF	0.880	0.939	0.443	0.635
	SVM	0.397	0.502	0.526	0.532
Galicaster	LR	0.722	0.778	0.524	0.471
	RF	0.877	0.938	0.519	0.526
	SVM	0.408	0.517	0.529	0.502
Lisa	LR	0.746	0.809	0.396	0.504
	RF	0.855	0.916	0.414	0.454
	SVM	0.390	0.503	0.435	0.439
Parsl	LR	0.702	0.769	0.309	0.528
	RF	0.843	0.924	0.316	0.506
	SVM	0.493	0.510	0.344	0.510
PyBitmessage	LR	0.688	0.746	0.376	0.521
	RF	0.832	0.916	0.376	0.405
	SVM	0.422	0.511	0.342	0.482
PythonRobotics	LR	0.743	0.841	0.496	0.361
	RF	0.819	0.908	0.497	0.424
	SVM	0.450	0.535	0.491	0.480
TADbit	LR	0.759	0.806	0.528	0.537
	RF	0.837	0.886	0.400	0.433
	SVM	0.573	0.520	0.550	0.556
Toil	LR	0.678	0.724	0.596	0.487
	RF	0.867	0.942	0.136	0.346
	SVM	0.602	0.526	0.125	0.485
Mean	**LR**	**0.708**	**0.768**	**0.476**	**0.494**
	RF	**0.850**	**0.921**	**0.433**	**0.499**
	SVM	**0.465**	**0.519**	**0.409**	**0.506**

commits, and scope of the projects. Nevertheless, the results may differ when our approaches are applied to commercial projects, larger or smaller systems. Future studies need to investigate whether our results generalize to other different projects. In addition, the results gained by using the automated tools for this experiment may vary according to their versions. Future work is necessary to analyze whether the same effect can be obtained on this research's approaches.

6.2 Internal Validity

The data for the independent and dependent variables that we used in this research relied on the dumped data of Commit Guru [31] and the processed results of the automated tools such as Radon [34] and PyDriller [32]. Although our results were concluded with several repeated experiments, the verification of the scripts of the automated tools and data was not performed in this research. Furthermore, we completed our performance results in precision, recall, F-measure, and AUC-ROC. Future studies, which have different objectives, should analyze our approaches' performance on other performance measures.

7. Conclusion

Just-in-time defect prediction approaches are practical and useful because of their ability to predict defects in the short-term and provide feedback immediately. However, a commit may contain multiple files, and a file may include many classes. For this reason, we proposed a commit-based class-level defect prediction approach for Python projects and analyzed our approach with two different settings. The main contributions of this research are:

1. A literature review about the most popular fine-grained defect prediction models for within-project area.
2. Commit-based class and file defect prediction models with two settings, and performance comparison for these settings.
3. Performance comparison for different classifiers and validation strategies for the commit-based class-level and file-level defect prediction models.

Our future work includes analyzing the reasons causing the worst performance for the commit-based class/file-level defect prediction model in a real-time scenario, replicating our study on a larger or smaller set of systems and industrial projects, and experimenting with different programming languages. Future studies can be conducted (i) to evaluate our study with different performance measures, (ii) to train and test with other metrics, (iii) to investigate the effort saved by using our study, and (iv) to apply in the context of cross-project defect prediction.

Acknowledgements

This research was partially supported by JSPS KAKENHI Japan (Grant Numbers: JP22K17874, JP18H04094, JP19K20240, and JP21H03416).

References

[1] S. Kim, E.J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" IEEE Trans. Softw. Eng., vol.34, no.2, pp.181–196, March–April 2008.
[2] Z. Wan, X. Xia, A.E. Hassan, D. Lo, J. Yin, and X. Yang, "Perceptions, expectations, and challenges in defect prediction," IEEE

Trans. Softw. Eng., vol.46, no.11, pp.1241–1266, Nov. 2018.
[3] Y. Kamei, E. Shihab, B. Adams, A.E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," IEEE Trans. Softw. Eng., vol.39, no.6, pp.757–773, June 2012.
[4] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," J. Syst. Softw., vol.150, pp.22–36, April 2019.
[5] E. Giger, M. D'Ambros, M. Pinzger, and H.C. Gall, "Method-level bug prediction," Proc. 2012 ACM-IEEE Int. Symp. Empir. Softw. Eng. Meas., pp.171–180, IEEE, Sept. 2012.
[6] A. Liaw and M. Wiener, "Classification and regression by randomforest," R news, vol.2, no.3, pp.18–22, Dec. 2002.
[7] P. McCullagh and J.A. Nelder, Generalized linear models, Routledge, 2019.
[8] C. Cortes and V. Vapnik, "Support-vector networks," Mach. Learn., vol.20, no.3, pp.273–297, Sept. 1995.
[9] P.A. Devijver and J. Kittler, Pattern recognition: A statistical approach, Prentice Hall, 1982.
[10] M. Kondo, D.M. German, O. Mizuno, and E.-H. Choi, "The impact of context metrics on just-in-time defect prediction," Empir. Softw. Eng., vol.25, no.1, pp.890–939, Jan. 2020.
[11] T. Lee, J. Nam, D. Han, S. Kim, and H.P. In, "Developer micro interaction metrics for software defect prediction," IEEE Trans. Softw. Eng., vol.42, no.11, pp.1015–1035, Nov. 2016.
[12] C. Pornprasit and C.K. Tantithamthavorn, "Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction," 2021 IEEE/ACM 18th Int. Conf. Mining Software Repositories (MSR), pp.369–379, IEEE, 2021.
[13] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto, "Predicting defective lines using a model-agnostic technique," IEEE Trans. Softw. Eng., vol.48, no.5, pp.1480–1496, May 2020.
[14] J. Nam, "Survey on software defect prediction," Department of Compter Science and Engineerning, The Hong Kong University of Science and Technology, Tech. Rep., 2014.
[15] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," 2012 34th Int. Conf. Softw. Eng. (ICSE), pp.200–210, IEEE, 2012.
[16] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," Information and Software Technology, vol.54, no.3, pp.248–256, March 2012.
[17] J. Nam, S.J. Pan, and S. Kim, "Transfer defect learning," 2013 35th Int. Conf. Softw. Eng. (ICSE), pp.382–391, IEEE, 2013.
[18] B. Turhan, T. Menzies, A.B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," Empir. Softw. Eng., vol.14, no.5, pp.540–578, Oct. 2009.
[19] S. Watanabe, H. Kaiya, and K. Kaijiri, "Adapting a fault prediction model to allow inter languagereuse," Proc. 4th International Workshop on Predictor Models in Software Engineering, pp.19–24, May, 2008.
[20] A. Perera, A. Aleti, M. Böhme, and B. Turhan, "Defect prediction guided search-based software testing," 2020 35th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE), pp.448–460, IEEE, Dec. 2020.
[21] F. Palomba, M. Zanoni, F.A. Fontana, A. De Lucia, and R. Oliveto, "Toward a smell-aware bug prediction model," IEEE Trans. Softw. Eng., vol.45, no.2, pp.194–218, Feb. 2017.
[22] A. Ahluwalia, D. Falessi, and M. Di Penta, "Snoring: A noise in defect prediction datasets," 2019 IEEE/ACM 16th Int. Conf. Mining Software Repositories (MSR), pp.63–67, IEEE, 2019.
[23] A. Agrawal and T. Menzies, "Is "better data" better than "better data miners"?: on the benefits of tuning SMOTE for defect prediction" 2018 IEEE/ACM 40th Int. Conf. Softw. Eng. (ICSE), pp.1050–1061, IEEE, May 2018.
[24] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A developer centered bug prediction model," IEEE Trans. Softw. Eng., vol.44, no.1, pp.5–24, Jan. 2017.
[25] Y. Qu, T. Liu, J. Chi, Y. Jin, D. Cui, A. He, and Q. Zheng, "node2defect: Using network embedding to improve software defect prediction," 2018 33rd IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE), pp.844–849, IEEE, Sept. 2018.
[26] S. Wang, J. Nam, and L. Tan, "Qtep: Quality-aware test case prioritization," Proc. 2017 11th Joint Meeting on Foundations of Software Engineering, pp.523–534, Aug. 2017.
[27] H. Osman, M. Ghafari, O. Nierstrasz, and M. Lungu, "An extensive analysis of efficient bug prediction configurations," Proc. 13th Int. Conf. Predictive Models and Data Analytics in Software Engineering, pp.107–116, Nov. 2017.
[28] Y. Koroglu, A. Sen, D. Kutluay, A. Bayraktar, Y. Tosun, M. Cinar, and H. Kaya, "Defect prediction on a legacy industrial software: A case study on software with few defects," 2016 IEEE/ACM 4th International Workshop on Conducting Empirical Studies in Industry (CESI), pp.14–20IEEE, May 2016.
[29] T. Diamantopoulos and A. Symeonidis, "Towards interpretable defect-prone component analysis using genetic fuzzy systems," 2015 IEEE/ACM 4th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, IEEE, 2015, pp.32–38.
[30] G. Nguyen, S. Dlugolinsky, M. Bobák, V. Tran, Á.L. García, I. Heredia, P. Malík, and L. Hluchỳ, "Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey," Artif. Intell. Rev., vol.52, no.1, pp.77–124, June 2019.
[31] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," Proc. 2015 10th Joint Meeting on Foundations of Software Engineering, pp.966–969, Aug. 2015.
[32] "Pydriller documentation!" [Online]. Available: https://pydriller.readthedocs.io/.
[33] "Ast - abstract syntax trees." [Online]. Available: https://docs.python.org/3/library/ast.html
[34] "Welcome to radon's documentation!" [Online]. Available: https://radon.readthedocs.io/.
[35] F. Akiyama, "An example of software system debugging." IFIP congress (1), vol.71, pp.353–359, 1971.
[36] M.H. Halstead, Elements of Software Science (Operating and programming systems series), Elsevier Science Inc., 1977.
[37] T.J. McCabe, "A complexity measure," IEEE Trans. Softw. Eng., vol.SE-2, no.4, pp.308–320, Dec. 1976.
[38] A. Agrawal, W. Fu, D. Chen, X. Shen, and T. Menzies, "How to "dodge" complex software analytics," IEEE Trans. Softw. Eng., vol.47, no.10, pp.2182–2194, Oct. 2019.
[39] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten, "The WEKA data mining software: an update," ACM SIGKDD Explorations Newsletter, vol.11, no.1, pp.10–18, Nov. 2009.
[40] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," IEEE Trans. Softw. Eng., vol.38, no.6, pp.1276–1304, Nov.–Dec. 2011.
[41] B. Wójcicki and R. Dabrowski, "Applying machine learning to software fault prediction," e-Informatica Software Engineering Journal, vol.12, no.1, pp.199–216, 2018.
[42] R. Widyasari, S.Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J.E. Tan, Y. Yieh, B. Goh, F. Thung, H.J. Kang, T. Hoang, D. Lo, and E.L. Ouh, "Bugsinpy: A database of existing bugs in python programs to enable controlled testing and debugging studies," Proc. 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp.1556–1560, Nov. 2020.

Khine Yin Mon is a Ph.D. Student at the Department of Information Science, Kyoto Institute of Technology. She received her M.E from Kyoto Institute of Technology and her B.C.Sc from University of Computer Studies, Yangon. Her research interests include Software Mining, Software Metrics, Software Quality Assurance, and Empirical Software Engineering. Contact her at k-yinmon@se.is.kit.ac.jp.

Masanari Kondo currently works as an assistant professor in the Principles of Software engineering and programming Languages Lab. (POSL) at Kyushu University, Kyushu, Japan. He was a Ph.D. student in the Software Engineering Laboratory (SEL) at the Kyoto Institute of Technology, Kyoto, Japan. He was also a Young Scientist of JSPS Research Fellowships (DC1) (2019.4–2021.3) and a visiting researcher in the Software Analysis & Intelligence Lab (SAIL) at Queen's University, Kingston, Canada (2018.3–2019.3). His research interests include supporting software developers by providing tools and methods for software quality assurance activities based on software repository mining, machine learning, and statistical analysis techniques. He received his BSc, MSc, and Ph.D. degrees from the Kyoto Institute of Technology (2017, 2019, and 2021). More about Masanari can be read on his website: https://mkmknd.github.io/

Eunjong Choi is a tenure-track assistant professor at Faculty of Information and Human Sciences in Kyoto Institute of Technology, Japan from Mar. 2019. Before joining Kyoto Institute of Technology, I was an assistant professor at Nara Institute of Science and Technology (NAIST) from Apr. 2016 to Mar. 2019 and at Osaka School of International (OSIPP), Osaka University from Apr. 2015 to Mar. 2016. She obtained Ph.D. of Information Science and Technology at the Graduate School of Information Science and Technology, Osaka University in 2015. Her research interests are in the areas of software engineering, in particular reused code management/detection, refactoring support, and test code generation.

Osamu Mizuno received M.E. and Ph.D. degrees from Osaka University in 1998 and 2001, respectively. Currently, he is a Professor at the Faculty of Information and Human Sciences, Kyoto Institute of Technology. His current research interests include software repository mining, fault-prone module prediction, software process improvement, and risk evaluation and prediction of software development. He is a member of IEEE and IPSJ.