Commit-based Class-level Defect Prediction for Python Projects

Thesis for the degree of Master of Engineering

Author:
Khine Yin Mon

Student ID No.:
20622009

Chief Supervisor:
Professor Osamu Mizuno

Co-Supervisor:
Assistant Professor Eunjong Choi

Master's Program of Information Science,
Graduate School of Science and Technology,
Kyoto Institute of Technology

February 7, 2022

様式 2 号

学位論文内容の要旨（和文）

令和 4 年 2 月 7 日

京都工芸繊維大学大学院
工芸科学研究科長　殿

工芸科学研究科　情報工学専攻
令和 2 年入学
学生番号　20622009
氏　名　Khine Yin Mon　㊞

（主任指導教員　Osamu Mizuno　㊞　）

本学学位規則第 4 条に基づき、下記のとおり学位論文内容の要旨を提出いたします。

1. 論文題目

Commit-based Class-level Defect Prediction for Python Projects

2. 論文内容の要旨（400 字程度）

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 proposed our novel class-level prediction, which is more fine-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 nine open-source python projects. We performed our experiment with two settings: setting with product metrics only and setting with product metrics plus commit information. We proved that applying commit information to the class-level prediction model can improve 30% of the performance in terms of AUC-ROC. Finally, we developed a commit-based file-level defect prediction model and compared it with the commit-based class-level defect prediction. Our findings reveal that the latter approach not only contributes to the fine-grained granularity but is also better in performance than the former one.

Commit-based Class-level Defect Prediction for Python Projects

2022 20622009 *Khine Yin Mon*

Abstract

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 proposed our novel class-level prediction, which is more fine-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 nine open-source python projects. We performed our experiment with two settings: setting with product metrics only and setting with product metrics plus commit information. We proved that applying commit information to the class-level prediction model can improve 30% of the performance in terms of AUC-ROC. Finally, we developed a commit-based file-level defect prediction model and compared it with the commit-based class-level defect prediction. Our findings reveal that the latter approach not only contributes to the fine-grained granularity but is also better in performance than the former one.

Index

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, e.g., testing and code reviews [3].

Defect prediction models predict the defect-prone parts of the software, and the size of these parts can be varied according to their prediction granularity. The defect prediction granularity exists from the finest token-level to the coarsest sub-system [4]. Many defect prediction approaches take the information from the past releases of the software and make predictions for future releases. These approaches are referred to as long-term prediction models [5]. The limitation of the long-term models is that they cannot provide immediate feedback to the developers. Hence, Kamei et al. introduced a Just-in-time defect prediction approach that can predict whether a commit can be a defect-introducing commit or not [3].

Pascarella et al. reported 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 [5]. Consequently, the file-level defect prediction, which is exploited by the commit information, is conducted. However, file-level defect prediction is still too coarse since the developers have to take a considerable amount of time to inspect all the codes in the entire file [6]. The more fine-grained level such as class-level and function-level than the file-level should be oriented for this approach. Moreover, our ultimate goal is to develop a more fine-grained defect prediction approach that uses the commit information.

To this aim, we first examine which granularity, i.e., which level defect prediction, is appropriate for our approach. We survey the popularity among fine-grained models, and our survey result led to choosing the class-level granularity to build our intended model.

Subsequently, we propose our novel commit-based class-level defect prediction approach and experiment with two settings: product metrics only approach and product metrics plus commit information approach. The comparison results of the two settings describe that the class-level defect prediction approach can improve by adding commit information. Finally, we build a commit-based file-level defect prediction approach and compare the results with the class-level approach. The results reveal that the class-level approach not only contributes to the fine-grained granularity but is also better in performance than the file-level one.

2. Related work

2.1 Defect prediction

Defect prediction approaches can be divided into two categories: long-term prediction approaches and short-term prediction approaches. 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 [3]. Meanwhile, short-term prediction approaches exploit the commit information's characteristics to predict the future defect-prone commits. The short-term prediction approaches predict at the change level and provide immediate feedback for the defect [5].

2.2 Defect prediction granularity

Widespread studies of defect prediction approaches are proposed based on machine learning techniques. The general process of machine learning-based defect prediction approaches includes generating instances from software archives, labeling these instances, preprocessing (optional), training these instances to become a model, and finally predicting for new instances by the trained model [4]. The mentioned instances are extracted from software archives such as version control systems, issue tracking systems, e-mail archives, etc. The size of these instances may vary according to their granularity [4]. The granularity can be described as a pyramid shape as mentioned in Figure 2.1 as a software system is constituted layer by layer. Therefore, each extracted or predicted instance can be a sub-system, component/ package, file, class, function, method, line, or 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 [6], [7].

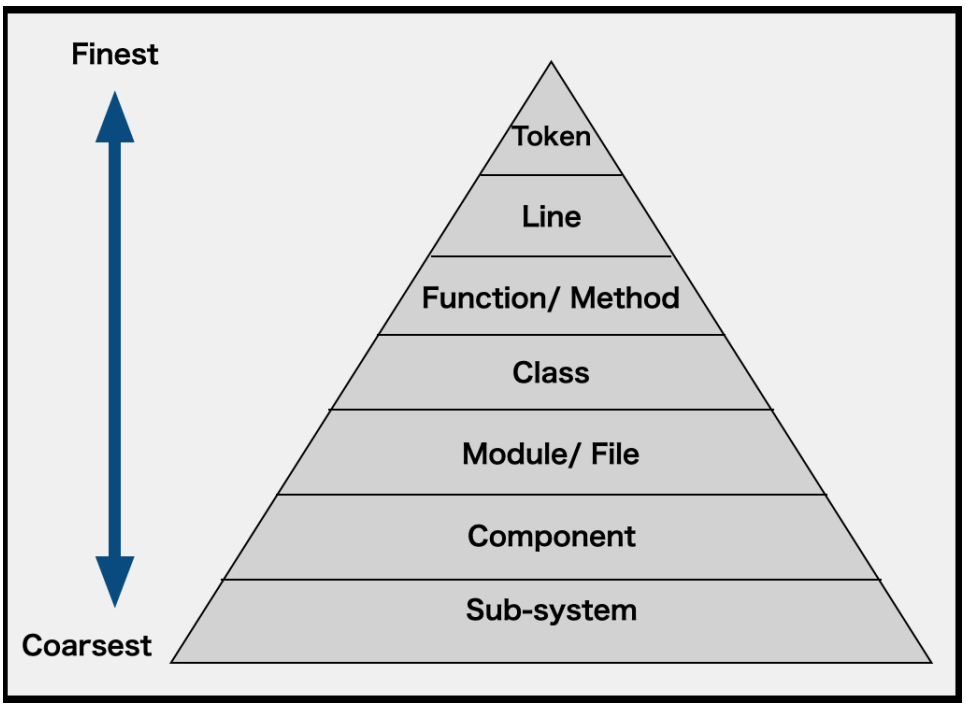

Figure 2.1 The granularity of defect prediction models

2.3 Defect prediction metrics

Product metrics and process metrics are commonly used for statistical defect prediction models. The product metrics (also known as code metrics) are collected directly from the source code. The popular product metrics are LOC [8], Halstead [9], McCabe [10], CK [11] and OO [12]. The process metrics are extracted from historical information archived in software repositories such as version control and issue tracking systems [4]. Relative code change churn [13], Change Entropy [14], and Popularity [15] are examples of process metrics. Besides product and process metrics, there are also other metrics such as anti-pattern metrics.

2.4 Commit

Modern software development relies on version control systems such as Git, CVS, SVN, etc. The version control systems allow every record of specific changes to the source code file. A commit or “revision” describes changes to a file or a set of files. (*1)

(*1): https://docs.github.com/

3. Methodology

This section presents our research questions, studied datasets, and outlines of our method.

3.1 Research questions

Our ultimate goal was to build a fine-grained defect prediction model which uses the commit information. Our proposed model design was a two-phase prediction model. In the first phase, we separated defective and non-defective commits. We identified and predicted the fine-grained defective artifacts in the second phase. However, choosing a reasonable fine-grained granularity was a challenge. For that reason, we decided to survey the popular granularity of fine-grained defect prediction models, and we set our first question as follows:

- **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 two-phase proposed model predict for a class of a commit?**

Prior research has already proved that the cost-effectiveness of the fine-grained defect prediction models overcomes that of the coarse-grained ones [6]. In this research, we developed a more coarse-grained granularity model than the granularity of class-level model and compared their performance results. Section 6 reveals the performance comparison results of two granularity levels.

- **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 Subject systems

We conducted our prediction model with open-source software systems. After applying multiple criteria, we removed many unwanted projects, and finally, nine projects which satisfied all criteria were randomly selected. Our criteria include that the projects :(1) must be publicly available, (2) must be available both in Commit Guru and GitHub, (3) should be mainly conducted with python or jupyter notebook language, (4) should have over 1000 commits, and (5) should include over 10% of defect-prone commits. The selected projects are listed in Table 3.1.

Table 3.1 Characteristics of the subject software systems.

Systems	# of Commits	% of Defect-prone Commits	# of Classes
ADSM	3493	24%	1516
Axelrod	5539	24%	6693
Bitmask_client	3055	18%	1241
Galicaster	1786	20%	360
Lisa	3876	18%	3511
Parsl	3724	32%	948
PyBitmessage	2595	30%	282
PythonRobotics	1700	20%	451
TADbit	2503	42%	66

3.3 Independent variables

Product metrics: We adopted LOC, Halstead, and Cyclomatic Complexity for our prediction models in this study. Table 3.2 reports the information of these adopted metrics. (*2)

Commit information Throughout this thesis, we mention the commit information used to build our proposed model. Just-in-time defect prediction models use the com-

(*2): https://radon.readthedocs.io/

mit information, and we used the following listed Kamei's Just-in-time defect prediction metrics [3] for this study. NS, ND, NF, Entropy, FIX, NDEV, AGE, NUC, EXP, and REXP. Although the original metrics include 13 variables, we exclude LT, LA, and LD as these three variables were not relevant to our current prediction model. The commit information adopted by this study is listed in Table 3.3.

3.4 Commit-based class-level defect prediction approach

According to research question 1, we built our proposed model as a commit-based class-level defect prediction model. Figure 3.1 illustrates the process flow of our approach. The overall steps of the process are listed below.

- We collected all the defective and non-defective commits.
- All the classes of the modified files of the commits were extracted, and these classes' product metrics were calculated.
- We made a dataset that contains both commit information and product metrics.
- Our model was trained and tested with the gained dataset.

(1) Data extaction

Our goal was to create a dataset that contains both commit information and product metrics of the classes. We gained the commit information (i. e., the calculated features of the commits) from Commit Guru. Moreover, the source code of the project's files and each file's revision history were obtained from GitHub repositories. The mutual commit hashes in the GitHub repository and Commit Guru were collected, and PyDriller took out the modified files of these commits. We extracted all modified files' classes using python's ast tool and calculated LOC, Halstead, and Cyclomatic Complexity metrics. Finally, a new dataset was acquired by concatenating the calculated metrics with the commit information. The dependent variable of our dataset is the availability of bugs in the classes, and independent variables are reported in Table 3.2 and Table 3.3.

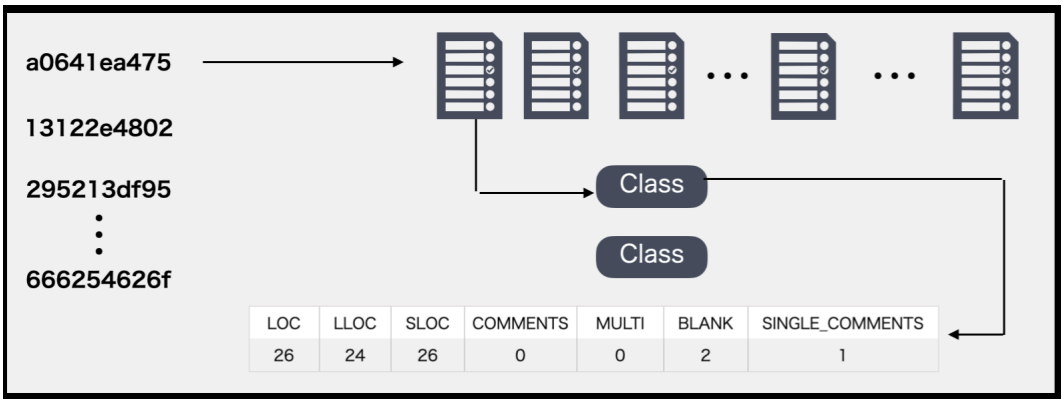

Figure 3.1 The process flow of our commit-based class-level defect prediction approach.

Tools for data extraction

Radon: Radon assists defect prediction by calculating the metrics such as LOC, Halstead, and Cyclomatic Complexity metrics. It is used as a python library or command-line tool. (*3)

PyDriller: PyDriller is a python frame that allows developers to mine the Git repositories with automated tools. With the help of PyDriller, the developers can extract Git repository ' s information such as commits, modified files, diffs, and source code, and export as CSV files easily. (*4)

(2) Model building and validation strategy

We trained our model with two different settings in the model building section. Our overall independent variables were 31 variables. 11 variables belonged to commit information obtained from Commit Guru data. The last 20 variables were the calculated product metrics. We trained our model by 31-attribut (commit information + product metrics) and 20-attribute (only product metrics) settings. There is no class-level defect prediction that exploits the commit information yet to the best of our knowledge. We assumed that the commit information would be enormously significant to the respective classes as we extracted the classes from the modified files of the commits. Once preparation of our data settings was done, we selected the best classier that supports predicted defective classes by our independent variables. For this purpose, we used Logistic Regression, J-48, Naive Bayes, and Random Forest as the same classifiers in the prior researches [3] [5]. We exploited the Weka toolkit for training our model. 10-fold cross-validation was applied as the validation strategy. 10-fold cross-validation strategy divides the dataset into ten folds. Nine folds are used as the training data, and the last fold is for testing. This procedure is receptive ten times, but a random 10-fold data partition is made for each time. Afterward, the accuracy result is taken as the mean value of the ten times validation.

(*3): https://radon.readthedocs.io/
(*4): https://pydriller.readthedocs.io

3.5 Commit-based file-level defect prediction approach

To answer research question 3, we built our file-level defect prediction first. For the sake of fairness, we performed our file-level prediction model with the same datasets, independent and dependent variables, model training, and validation methodology approaches.

(1) Data extraction

We first dumped the Commit Guru data and GitHub repositories to build our file-level prediction model. Using PyDriller, we extracted the modified files, available in the dumped GitHub repository, with their source code and calculated the LOC, Halstead, and Cyclomatic Complexity metrics. Afterward, these metrics were combined with commit information from Commit Guru data and created as a new dataset.

(2) Model building and validation strategy

With the same model training method and validation strategy in the above section, we used the Random forest algorithm and 10-fold cross-validation. Finally, we resulted our output in precision, recall, F-measure, and AUC-ROC.

Table 3.2 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 3.3 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
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

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

4.1 Motivation

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 using the following method.

4.2 Approach

We defined the area scope of our literature review according to research question 1. Since this study was not intended to become a systematic literature review for a wide area of defect prediction research field, we only focused on surveying the granularity occurrence of the fine-grained defect prediction models. In Pascarella et al. ' s approach [5], they set file-level as the fine-grained granularity, and we aimed to improve their approach by developing a more fine-grained prediction model than the one of their model. Therefore, we counted for class, function, method, line, and token levels, that arefiner 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 and collected the titles of all resulting papers. Then we filtered the relevant papers by reading the titles, abstract, and keywords. 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], [17], [18], [19], and deep learning techniques. Finally, we downloaded the full text of the rest papers and found out the granularity of the predicted part.

4.3 Result

We observed ten class-level papers, two function-level papers, two method-level papers, and one line-level paper. As we discovered the most contributed granularity of defect prediction models, we recognized class-level granularity as the answer to our first research question. Furthermore, we built our proposed model for class-level defect prediction granularity. Figure 4.1 shows the result of research question 1. Among the empirical literature studies, 66.7% are class-level, 13.3 % are function-level, 13.3 % are method-level, and 6.7% are line-level. Therefore the class-level becomes the result of research question 1.

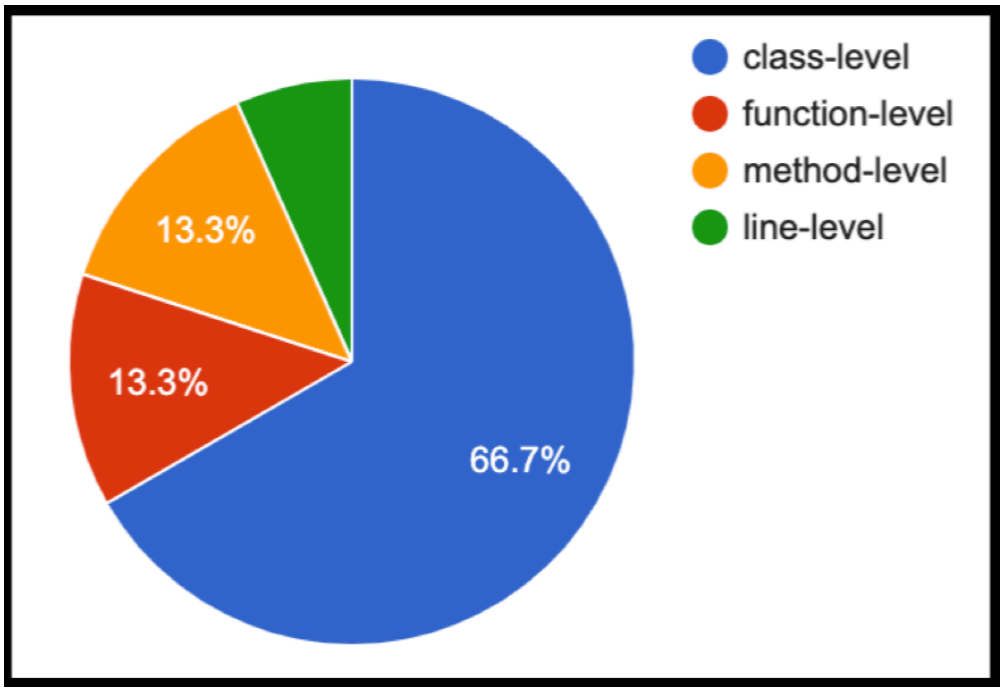

Figure 4.1 66.7% belong to class, 13.3% belong to function, 13.3% belong to method, and 6.7% belong to line levels.

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

5.1 Motivation

In 2019, Pascarella et al. proposed a fine-grained just-in-time defect prediction model [5]. Their model classifies and predicts the non-defective and defective files based on the defective commits. Although their model is orienting to the fine-grained granularity, the file-level prediction might be too coarse as a file may contain numerous classes. For this reason, we proposed a commit-based class-level defect prediction approach, in which defect-prone classes are generated from the commits.

5.2 Approach

To answer our research question 2, we conducted a commit-based class-level defection prediction approach. We generated independent and dependent variables metrics from the Commit Guru dumped data and GitHub repositories. Our approach was tested by two settings: (1) product metrics only (20-attribute setting) and (2) product metrics plus commit information (31-attribute setting). The product metrics contained LOC, Halstead, and Cyclomatic Complexity metrics. The commit information refer to the number of modified subsystems (NS), the number of modified directories (ND), number of modified files (NF), Entropy & Distribution of modified code across each file (Entropy), whether or not the change is a defect fix (FIX), the number of developers that changed the modified files (NDEV), the average the time interval between the last and the current change (AUE), number of unique changes to the modified files (NUC), developer experience (EXP), recent developer experience (REXP), and developer experience on a subsystem (SEXP). We trained and tested our model with various classifier and validation strategies. Among these, the Random Forrest classifier and 10-fold cross-validation gave our approach the best performance.

5.3 Result

We evaluated the performance measures of our prediction model, which was trained and tested by the Random Forest classier and 10-fold cross-validation, as described in section 3. Table 5.1 and Table 5.2 provide precision, recall, F-measures, and AUC-ROC percentage of our results. For the sake of clarity, we present the results of two settings with separate tables that have a similar structure.

We performed our predictions only with the product metrics (20-attribute setting) and presented the results in Table 5.1. The precision ranged from 0.544 to 0.695 (average = 60%), indicating our model could give 60 % reliability for predicting actual defects. The recall ranged from 0.559 to 0.697 (average = 62%). According to its recall value, our model could predict more than 60% of the actual defects. The average F-measure was 60%, achieving an AUC-ROC range of 54-83% (average = 62%). Although the results fluctuated significantly, our model still gave the average performance for various projects.

Table 5.2 describes our predictor's performance result, which uses the combined features for commit information and product metrics (31-attribute setting). We observed that the ranges of precision and recall were between 82% to 97% (average = 91%) while F-measure ranged from 73% to 96% (average = 73%). The AUC-ROC was between 95% to 99%, and the average was 95%. This is proof that commit information can contribute greatly to the class-level defect prediction model.

5.3.1 Comparison for 20-attribute and 31-attribute settings for the class-level defect prediction approach

The precision and recall of the 31-attribute setting were 11% higher than that of the 20-attribute setting. The F-measure and AUC-ROC values were 9% and 8% greater in the 31-attribute setting, respectively. Therefore, we concluded that adding commit information to the product metrics for our class-level defect prediction approach can improve its performance, and commit information significantly contributes to the commit-based class-level defect prediction approach.

Table 5.1 Performance result of the class-level prediction model (20-attribute setting).

Systems	Precision	Recall	F-measure	AUC-ROC
ADSM	0.695	0.697	0.693	83%
Axelrod	0.567	0.585	0.558	57%
Bitmask_client	0.544	0.559	0.543	54%
Galicaster	0.557	0.560	0.557	59%
Lisa	0.668	0.674	0.642	66%
Parsl	0.586	0.595	0.587	60%
PyBitmessage	0.822	0.837	0.824	82%
PythonRobotics	0.567	0.574	0.565	61%
TADbit	0.637	0.658	0.644	66%

Table 5.2 Performance result of the class-level prediction model (31-attribute setting).

Systems	Precision	Recall	F-measure	AUC-ROC
ADSM	0.960	0.960	0.960	99%
Axelrod	0.967	0.967	0.932	99%
Bitmask_client	0.954	0.953	0.953	90%
Galicaster	0.853	0.853	0.852	91%
Lisa	0.954	0.952	0.899	99%
Parsl	0.870	0.870	0.730	95%
PyBitmessage	0.969	0.969	0.969	99%
PythonRobotics	0.854	0.854	0.854	93%
TADbit	0.820	0.824	0.817	89%

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

6.1 Motivation

One of the current trends in the defect prediction research area is developing more fine-grained models because of the advantages of performing predictions at a finer granularity [20]. In this study, we conducted two fine-grained granularity prediction models - the class-level and the file-level - and compared their performance results.

6.2 Approach

To answer our research question 3, we need to present the performance result of the file-level defect prediction model first. For both of the class-level and file-level models, we extracted the metrics from Commit Guru dumped data and Github repositories. Although the calculation of the metrics was slightly different due to their granularity, we used the same machine learning classifier, validation strategy, and performance measures.

Similar to the setting of the class-level defect prediction model, we tested the file-level defect prediction in two settings: (1) product metrics only (20-attribute setting) and (2) product metrics plus commit information (31-attribute setting). The number of the utilized product metrics was 20, and they were LOC, Halstead, and Cyclomatic Complexity metrics. The metrics of commit information included number of modified subsystems (NS), number of modified directories (ND), number of modified files (NF), Entropy & Distribution of modified code across each file (Entropy), whether or not the change is a defect fix (FIX), the number of developers that changed the modified files (NDEV), the average time interval between the last and the current change (AUE), number of unique changes to the modified files (NUC), developer experience (EXP), recent developer experience (REXP), and developer experience on a subsystem (SEXP). The commit information

was the metrics applied in the state-of-art Just-in-time defect prediction study by Kamei. et al. [3]. Although the original study stated 13 metrics for commits, we excluded the size metrics, which are lines of code added (LA), lines of code deleted (LD), and lines of code in a file before the change (LT) as they are not relevant to our approaches.

6.3 Result

Table 6.1 and Table 6.2 describe the performance results of the commit-based file-level defect prediction model trained by the Random Forest classifier with 10-Fold cross-validation. We present the results in terms of precision, recall, F-measure, and AUC-ROC score. The results involved in Table 6.1 were gained by training the model using only product metrics (20-attribute setting). The precision value ranged from 0.546 to 0.651 (average = 0.57), the recall value was from 0.547 to 0.653 (average = 0.58). The range of the F-measure was from 0.546 to 0.652, and its average was 0.58. We achieved AUC-ROC score values between 0.56 and 0.729, and the average AUC-ROC value was 0.6 for the 20-attribute setting.

Table 6.2 presents the precision, recall, F-measure, and AUC-ROC values obtained by the product metrics plus the commit information (31-attribute setting). When we used the 11 features of commit information for our product metrics only file-level defect prediction model, we obtained the precision ranging from 0.75 to 0.838 (average = 0.801), the recall ranging from 0.752 to 0.837 (average = 0.801), the F-measure ranging from 0.75 to 0.837 (average = 0.799), and AUC-ROC ranging from 0.816 to 0.92 (average = 0.873).

6.3.1 Comparison for 20-attribute and 31-attribute settings for the file-level defect prediction approach

When we used the 11 features of commit information for our product metrics only file-level defect prediction model, the precision and the recall increased for 23% and 22% respectively. F-measure of 31-attribute setting became 21% greater than that of 20-attribute setting, and AUC-ROC score improved for 27%. Since our file-level defect

Table 6.1 Performance result of the file-level prediction model (20-attribute setting).

Systems	Precision	Recall	F-measure	AUC-ROC
ADSM	0.651	0.653	0.652	73%
Axelrod	0.548	0.551	0.549	56%
Bitmask_client	0.547	0.550	0.548	56%
Galicaster	0.548	0.549	0.548	59%
Lisa	0.646	0.647	0.646	70%
Parsl	0.591	0.595	0.593	60%
PyBitmessage	0.561	0.562	0.561	57%
PythonRobotics	0.546	0.547	0.546	60%
TADBit	0.578	0.581	0.580	61%

prediction was gained from the modified files of the commits, we successfully proved our hypothesis that commit information greatly contributes to the commit-based file-level defect prediction.

6.3.2 Comparison result for the commit-based class-level and file-level defect prediction models

For the 31-attribute setting, we present the performance comparison with Figure 6.1, Figure 6.2, Figure 6.3, and Figure 6.4. The performance of the class-level model is 8-9% better than that of the file-level model in terms of F-measure and AUC-ROC. The class-level model ' s precision and recall values were 11% greater than the file-level model ' s ones.

According to the testing results with nine python projects, we concluded that the performance of the commit-based class-level prediction is up to 8% better than the commit-based file-level prediction model in terms of F-measure and AUC-ROC.

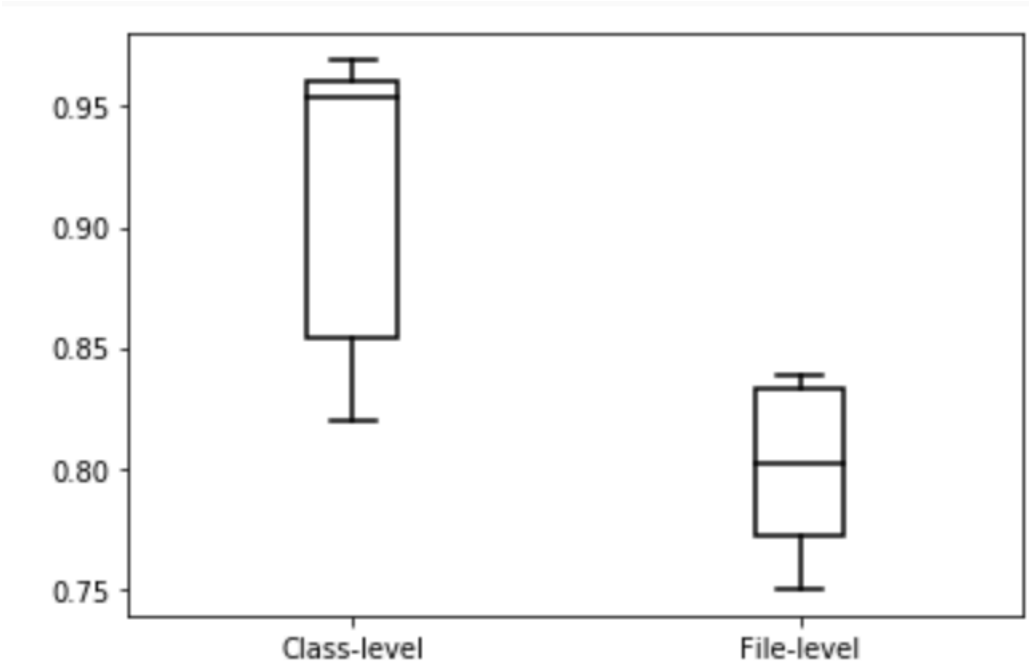

Figure 6.1 Comparison of precision boxplots for the commit-based class-level and file-level defect prediction models.

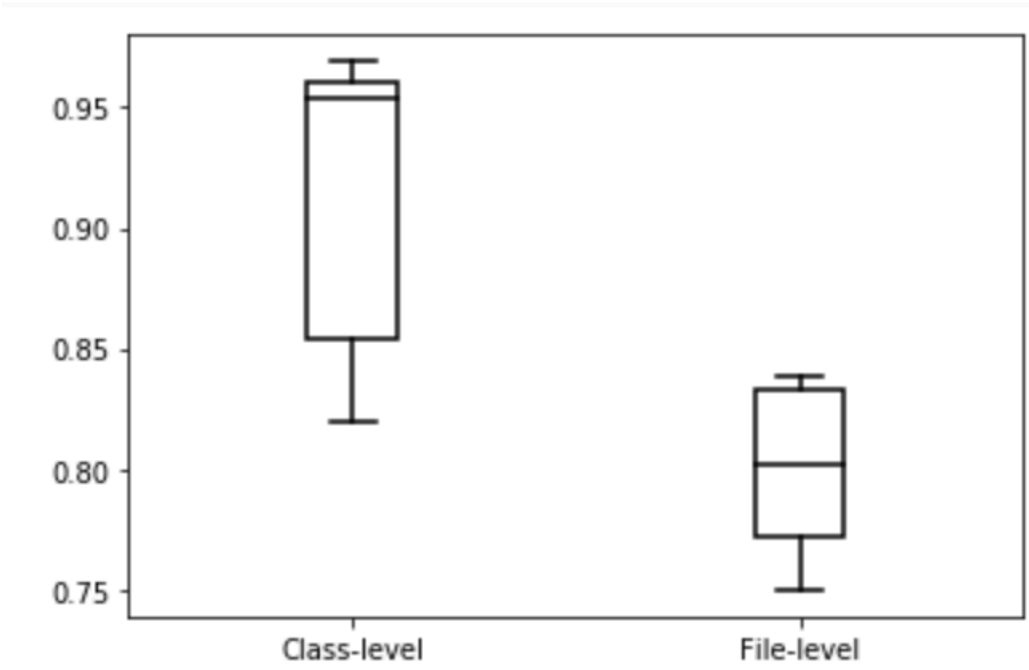

Figure 6.2 Comparison of recall boxplots for the commit-based class-level and file-level defect prediction models.

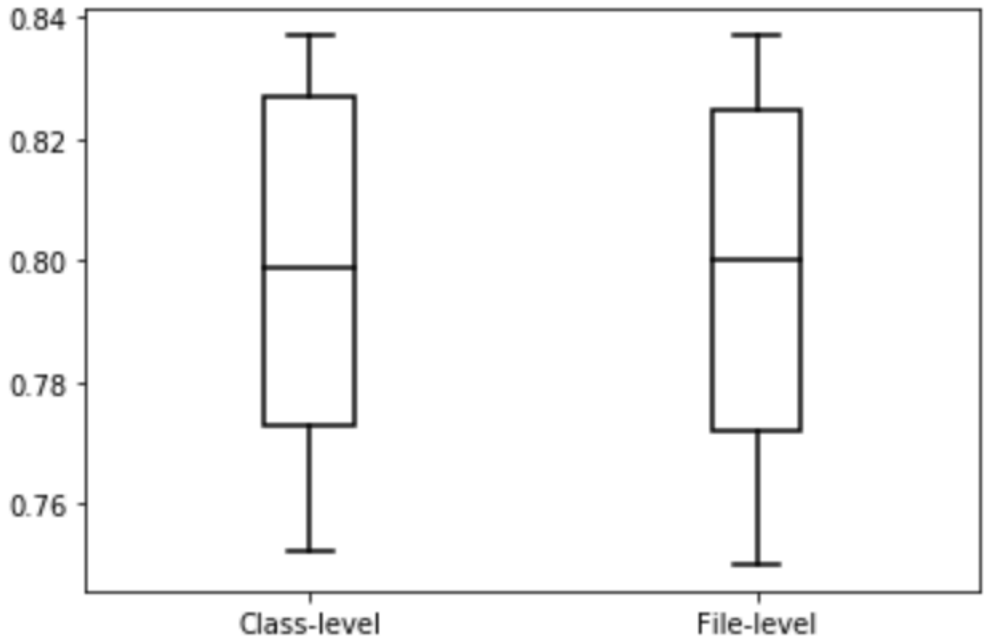

Figure 6.3 Comparison of F-measure boxplots for the commit-based class-level and file-level defect prediction models.

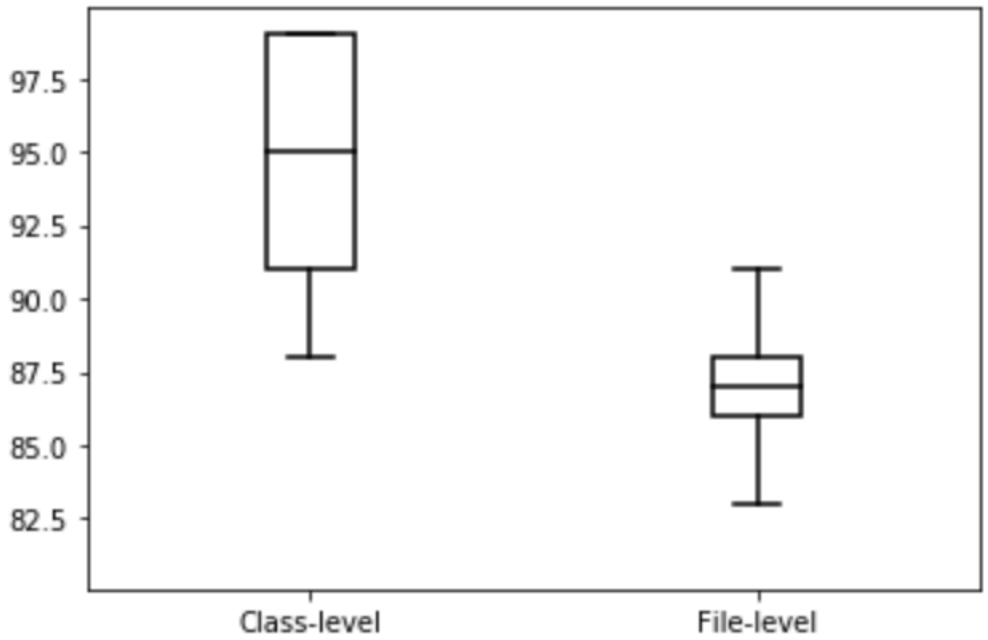

Figure 6.4 Comparison of AUC-ROC boxplots for the commit-based class-level and file-level defect prediction models.

Table 6.2 Performance result of the file-level prediction model (31-attribute setting).

Systems	Precision	Recall	F-measure	AUC-ROC
ADSM	0.750	0.752	0.750	84%
Axelrod	0.838	0.837	0.837	92%
Bitmask_client	0.823	0.822	0.821	88%
Galicaster	0.838	0.836	0.836	91%
Lisa	0.833	0.827	0.825	87%
Parsl	0.793	0.794	0.793	87%
PyBitmessage	0.772	0.773	0.772	86%
PythonRobotics	0.802	0.799	0.800	89%
TADBit	0.761	0.769	0.761	82%

7. Thread to validity

7.1 External validity

We experimented with our approaches with nine open-source python projects with different ratios of defect-prone commits, number of commits, and scope. Nevertheless, the results may differ when our approach is 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.

7.2 Internal validity

The data for the independent and dependent variables that we used in this research relied on the dumped data of Commit Guru and the processed results of the automated tools such as Radon and PyDriller. Although our results were concluded with several repeated experiments, the verification of the scripts of the tools and data was not performed in this research. Furthermore, we concluded 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.

8. 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 survey for the granularity of fine-grained defect prediction approaches.
2. A commit-based class-level defect prediction model with two settings and performance comparison for these settings.
3. A commit-based file-level defect prediction model with two settings and performance comparison for these settings.
4. Performance comparison for the commit-based class-level and file-level defect prediction models.
5. Our concluded results state that commit information significantly contributes to our commit-based class-level prediction approach, and the commit-based class-level model is up to 8% better than the commit-based file-level prediction model in terms of F-measure and AUC-ROC.

Acknowledgment

This research was possible to be conducted because of the continued support and engagement of many grateful people. I would like to express my deep and sincere gratitude to my research supervisor, Dr. Osamu Mizuno, for letting me do this research and providing invaluable support throughout this journey. To Dr. Eunjong Choi, for her trust, offering valuable advice, giving support during this whole study period, and encouraging me to finish the thesis. To Dr. Masanari Kondo, for sharing his expertise by giving constructive comments and suggestions upon reviewing this study.

References

[1] S. Kim, E.J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?," IEEE Transactions on Software Engineering, vol.34, no.2, pp.181–196, 2008.

[2] Z. Wan, X. Xia, A.E. Hassan, D. Lo, J. Yin, and X. Yang, "Perceptions, expectations, and challenges in defect prediction," IEEE Transactions on Software Engineering, vol.46, no.11, pp.1–26, 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 Transactions on Software Engineering, vol.39, no.6, pp.757–773, 2012.

[4] J. Nam, "Survey on software defect prediction," Department of Compter Science and Engineerning, The Hong Kong University of Science and Technology, Tech. Rep, 2014.

[5] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," Journal of Systems and Software, vol.150, pp.22–36, 2019.

[6] E. Giger, M. D'Ambros, M. Pinzger, and H.C. Gall, "Method-level bug prediction," Proceedings of the 2012 International Symposium on Empirical Software Engineering and Measurement, pp.171–180, 2012.

[7] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," Proceedings of International Conference on Software Engineering (ICSE), pp.200–210, IEEE, 2012.

[8] F. Akiyama, "An example of software system debugging.," IFIP Congress (1), pp.353–359, North-Holland, 1971.

[9] M.H. Halstead, Elements of Software Science (Operating and Programming Systems Series), Elsevier Science Inc., USA, 1977.

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

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

[12] F.B. eAbreu and R. Carapuça, "Candidate metrics for object-oriented software within a taxonomy framework," Journal of Systems and Software, vol.26, no.1, pp.87–96, 1994.

[13] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," Proceedings of the 27th International Conference on Software Engineering, pp.284–292, 2005.

[14] A.E. Hassan, "Predicting faults using the complexity of code changes," Proceedings of the IEEE 31st International Conference on Software Engineering, pp.78–88, IEEE, 2009.

[15] A. Bacchelli, M. D'Ambros, and M. Lanza, "Are popular classes more defect prone?," Proceedings of the International Conference on Fundamental Approaches to Software Engineering, pp.59–73, Springer, 2010.

[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, 2012.

[17] J. Nam, S.J. Pan, and S. Kim, "Transfer defect learning," Proceedings of the 35th International Conference on Software Engineering (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," Empirical Software Engineering, vol.14, no.5, pp.540–578, 2009.

[19] S. Watanabe, H. Kaiya, and K. Kaijiri, "Adapting a fault prediction model to allow inter languagereuse," Proceedings of the 4th International Workshop on Predictor models in software engineering, pp.19–24, 2008.

[20] Y. Kamei and E. Shihab, "Defect prediction: Accomplishments and future challenges," Proceedings of the IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER), vol.5, pp.33–45, IEEE, 2016.