Just-in-Time Defect Severity Prediction

Ran Mo, Yushuo Wang, Yao Zhang, Zengyang Li

School of Computer Science & Hubei Provincial Key Laboratory of Artificial Intelligence and Smart Learning
Central China Normal University
moran@ccnu.edu.cn, yswang@mails.ccnu.edu.cn, mm1314955811@mails.ccnu.edu.cn, lzy@ccnu.edu.cn

***Abstract*—To efficiently fix defects within a specific time frame during software development, researchers have proposed defect severity prediction to help developers determine which defects to fix first and make efficient use of limited resources. Additionally, to improve the efficiency of defect fixing, just-in-time (JIT) defect prediction has been proposed to promptly predict code fragments that may introduce defects when developers make code changes (i.e., submit a commit). In this way, defect feedback is prompt and localization precision is high. Typically, high-priority defects must be addressed as soon as possible, but when the bug report records a defect back to developers, they need to take time to get reacquainted with the related code fragments, slowing down the speed of high-priority defect fixes. Therefore, we used three machine learning algorithms to develop a JIT defect severity prediction model that allows developers to classify the severity of potential defects when submitting code changes. Our models were tested on ten large-scale projects and showed they can effectively predict defect severity just in time. With Random Forest, our models achieved an average precision of 0.552, and an average recall, F1-measure, and AUC of 0.579, 0.528, and 0.729, respectively. Using Decision Tree, the average precision, recall, F1-measure, and AUC achieve 0.479, 0.494, 0.485, and 0.619, respectively; Using KNN, the average precision, recall, F1-measure, and AUC are 0.466, 0.468, 0.467, and 0.593, respectively. Meanwhile, we find a large portion (90.3% on average) of defect-introducing changes are at a high severity level.**

***Index Terms*—JIT Defect Severity Prediction; Defect-fixing Changes; Defect-introducing Changes**

I. INTRODUCTION

During software development and maintenance, repairing software defects is an indispensable task, and it often needs to be completed within a specified time period [1]. Hence developers usually assign a severity level to different defects, so that developers can give priority to fixing the high-severity defects when given limited time [1], [2]. In addition, the defect severity level also helps to reasonably assign defect-repairing tasks to the developers with appropriate development experience, which could avoid incorrect repairs [3]. However, classifying defect severity is mainly done manually by testers relying on their own expertise and experience [2], [4]. For software systems with massive defects, the manual assessment of defect severity would be time-consuming and error-prone [4], [5]. Therefore, more and more researchers have paid attention to defect severity prediction and explored approaches for automatically predicting the severity of defects [1]–[7].

JIT defect prediction has become an active research area [8]–[11], which predicts code fragments that may introduce defects when developers make a code change (i.e. submit a commit) [8]. Compared with the defect prediction based on the coarse-grained level of modules, packages, files, etc., this change-level prediction can narrow the scope of the location of the defects introduced [8], so that developers can quickly review potential defects, and each change has a unique author [8], so it is easy to determine the developers who introduced the defects. In addition, the defects are predicted before the changes are submitted, so developers can immediately repair the code when they are very familiar with the code fragments [8], saving time in recalling the implementation decisions.

In this paper, we introduced the concept of JIT defect prediction into the prediction of defect severity. Unlike traditional methods, a defect often got documented and classified several weeks or months later than it was introduced. Consequently, developers may need to take time to get reacquainted with the related code fragments for analyzing or even fixing the defect. With the JIT prediction on defect severity, developers would be able to promptly classify the severity of potential defects when committing code changes. This would help developers quickly assess the severity of defects and allocate resources for further repairs. Especially for the high-severity defects, which need to be fixed as soon as possible, developers can quickly review code snippets and fix them, since they are familiar with the code, which saves time in recalling the code and speeds up the repair of high-priority defects.

In this study, we use Random Forest, Decision Tree, and KNN classifier to establish JIT defect severity prediction model and apply the model to ten large-scale projects. The results show that our models can effectively predict defect severity just in time. The Random Forest-based model can get a precision of 0.552 on average, and the average recall, F1-measure, and AUC are 0.579, 0.528, and 0.729, respectively. For the Decision Tree-based model, the average precision, recall, F1-measure, and AUC are 0.479, 0.494, 0.485, and 0.619, respectively. When using KNN, the average precision, recall, F1-measure, and AUC can reach 0.466, 0.468, 0.467, and 0.593, respectively. In addition, we find a large portion (90.3% on average) of defect-introducing changes are at a high severity level.

The rest of the paper is structured as follows: Section II introduces the background of this paper. Section III reports the related work. Section IV introduces the methodology of our model building. Section V presents the experimental findings. Section VI describes the shortcomings of this research and future work. Section VII concludes.

DOI reference number: 10.18293/SEKE23-218

II. Background

A. Bug Report

During software development and maintenance, issue tracking systems are often used to record and track possible change requests, such as feature addition, bug fix, improvement, etc., in which the bug-fixing information is recorded in the bug report. The format of bug reports may vary from the issue tracking system, but there is always some common information in a bug report, including the ID, description, severity of a bug, and the products or components affected by the bug [2].

B. Defect Severity

The defect severity level is used to characterize the degree of negative impact by software defects [1]. Commonly used issue tracking systems always have a field to denote the severity of a defect in the report. This paper describes the severity level through the *Priority* field provided by Jira [1]. We consider *Blocker*, *Critical*, and *Major* as high severity levels similar to [4], while *Minor* and *Trivial* are considered low severity levels.

C. Defect-fix change and Defect-introducing change

A defect-fix change is to fix a defect as recorded in the issue tracking system. Following the prior studies [12], we matched the committed changes with bug reports to identify defect-fix changes: if a commit's message references a bug id, we considered it to be a defect-fix change. A defect-introducing change is a change that introduces a defect that needs to be fixed in the future during a code change [13], [14]. In this paper, we identify defect-introducing changes by using the SZZ algorithm Section II-D elaborated in the next section.

D. SZZ Algorithm

Śliwerski et al. [13] first proposed the SZZ algorithm to automatically identify changes as defect-introducing or non-defect-introducing, which has played a critical role in JIT defect prediction. This algorithm mainly consists of these steps: 1) matching the bug reports with the commits recorded in revision history to identify defect-fix commits; 2) using the *diff* command of the version control system to compare a defect-fix commit with its previous commit to get the changed code fragments, i.e, defect-fix code; 3) using the *annotate* command to retrace the history of defect-fix code fragments to locate the nearest commit where the code was added. Then this commit is considered as the commit inducing the defect-fix code, thus it is a candidate for the defect-introducing commit (change); 4) filtering out the candidates whose defect-introducing time is later than the defect-fix time. Since a defect should be introduced first, then it can be fixed later [13].

III. Related Work

Our work is mainly related to JIT defect prediction and defect severity prediction. We will introduce some prior studies from these two aspects.

JIT defect prediction. Kamei et al. [8] applied SZZ algorithm [13] to mark code changes as defect-introducing or non-defect-introducing, extracted 14 metrics from the dimensions of *Diffusion*, *Size*, *Purpose*, *History*, and *Experience* and built a logistic regression model for JIT defect prediction, which showed 68% accuracy and 64% recall. Yang et al. [10] extracted the same features as Kamei et al. [8] from changes with known labels and employed a two-layer ensemble learning approach TLEL for JIT defect prediction. The result showed that TLEL was more effective than employing a single machine learning algorithm and it requires only checking 20% of the lines of code to discover over 70% of the defects. Borg et al. [15] utilized the SZZ algorithm [13] to label code changes, extracted metrics from various dimensions, such as *Diffusion*, *Size*, *Experience*, etc, and built a random forest model for JIT defect prediction. The researchers also made the SZZ algorithm open-source. In addition to the above supervised learning algorithms, researchers [16]–[18] have leveraged unsupervised learning algorithms to predict defects just in time without labeling data.

Defect severity prediction. Lamkanfi et al. [6] compared Naive Bayes, Naive Bayes Multinomial, Support Vector Machines, and K-Nearest Neighbour in severity prediction, and presented that Naive Bayes Multinomial is the most effective for classifying severe and non-severe bug reports in two open-source systems, Eclipse and GNOME. Sahin and Tosun [7] found that using word embeddings for feature extraction and constructing models using Convolutional Neural Networks (CNN), Long Short Term Memory (LSTM), and Extreme Gradient Boosting (XGBoost) algorithms can effectively predict defect severity. Meanwhile, they found that word embeddings and deep learning techniques can also be used to directly predict the severity score of defects. Arokiam et al. [5] utilized previous bug report writing styles as features to predict defect severity, which outperformed existing keyword-based methods and can detect defects early in new projects.

In recent years, researchers have proposed various studies for JIT defect prediction or defect severity prediction, but none of them has focused on JIT defect severity prediction. Our paper is the first attempt to address this research gap. When developers submit changes, it is helpful to predict which change may introduce defects and how severe the defects are. Thus we conduct JIT defect severity prediction models in this work.

IV. Methodology

This section focuses on our experimental approach. The flow chart of our method is shown in Fig. 1. It mainly includes the steps of labeling, feature extraction, and model building.

A. Labeling

In this paper, we used the tool of SZZ Unleashed [2] by Borg et al. [15] to obtain the commit Ids of all defect-fix and defect-introducing change pairs. Then we traversed all

[1] https://www.atlassian.com/software/jira

[2] https://github.com/wogscpar/SZZUnleashed

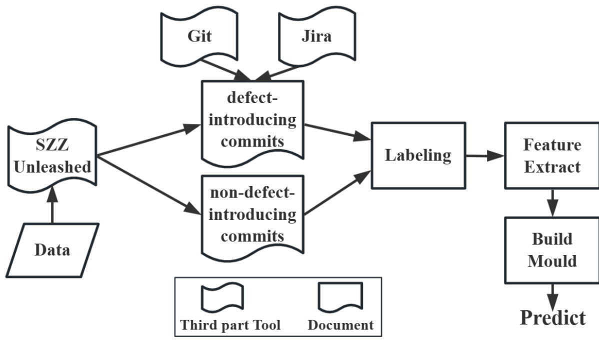

Fig. 1. Overview of our approach

TABLE I
EXTRACTED FEATURES

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

commits in a project's revision history to identify the defect-introducing commits: if a commit's id is not in the identified Ids, we label this commit as 0, which means that this commit does not introduce defects. For a defect-introducing commit, we first get its paired defect-fix commit. Then based on the bug Id in the defect-fix commit's message to analyzed the report of this defect to determine its severity. Following the prior studies, *Minor* and *Trivial* are considered to be low severity levels [4], since there is little difference in severity between them, thus we labeled both of them to be 1, *Major* is labeled as 2, *Critical* is labeled as 3, and *Blocker* is labeled as 4. The greater the value, the more severe the defect. If a commit introduced multiple defects, we labeled it based on the defect with the greatest severity.

B. Feature Extraction

Table I lists the features we extracted to represent each change. The first column shows the types of selected features; the second column is each feature; the third column shows the description of each feature. These fourteen features of the five dimensions used in this paper are widely used in JIT defect prediction [8]–[11].

Size dimension measures the size of the change. Larger code modifications are more likely to introduce defects, and relative code churn metrics are better defect predictors than absolute metrics [19], so we standardize the LA, LD, and LT by dividing LA and LD by LT and dividing LT by NF, as in [15].

Diffusion is used to measure the distribution of changed code in related files. When code change is spread across multiple related files, it can increase complexity and introduce potential defects [20]. We calculate these four features using the method described in [8].

Purpose takes the purpose of the modified code fragments into consideration. Defects are more likely to be introduced in the process of repairing [13], so many studies [9]–[11], [21] have used this type of feature for JIT defect prediction. For the FIX, we define its value based on whether the current change fixes the defect and the severity of the defect being fixed. If the change is not committed to fix the defect, FIX is 0. Otherwise, when the severity of the defect repaired by the change is *Minor* or *Trivial*, *Major*, *Critical* and *Blocker*, the Fix is 1, 2, 3, and 4, respectively.

History dimension measures the modification history of files that the current change is about to modify. The more times related files are modified or modified by more developers or the smaller the time difference between the latest change and this change in a relevant file, the more likely there are defects [8], [22]. Therefore, this dimension is commonly used for JIT defect prediction [9], [11], [15].

Experience dimension measures the experience of developers making changes. The more experienced the developer who changes the code, the less likely it is to introduce defects [21], so we use the features from this dimension for JIT defect prediction, similarly to [10], [11], [23].

C. Model Development

Machine Learning Algorithms. We selected three machine learning algorithms for building our prediction models. All of them have been widely used in defect predictions and support multi-classifications, with their effectiveness having been empirically validated [10], [15], [24], [25]: 1) Random Forest [26] is a classifier containing multiple decision trees. It can be used to build binary or multi-classification prediction models with high accuracy, robustness, and stability; 2) Decision Tree is a model that uses a tree-like data structure to represent decision rules and classification results. It can quickly find the characteristic variables that distinguish different categories and is suitable for multi-classification prediction; 3) k-Nearest Neighbor (KNN) is a method that classifies instances into K classes with high accuracy. It is one of the simplest machine learning algorithms and is suitable for multi-classification prediction.

In this paper, we used the Random Forest, Decision Tree (J48), and KNN classifiers that come with the Weka [3] toolkit to predict the severity of defects introduced by changes. We directly applied the default setting of each classifier in Weka in our prediction models.

Model Training and validation. Numerous studies of JIT defect prediction have used the ten-fold cross-validation method

[3]https://www.cs.waikato.ac.nz/ ml/weka/

to train and validate prediction models [18], [27]. The ten-fold cross-validation method first disrupts the dataset and divides it into ten parts. Then it selects one of them as the test set and the remaining nine as the training set. A total of ten experiments are run, and the average of the ten results is taken as the evaluation result of the ten-fold cross-validation. The advantage of this method is that each set of data can be used as training data and test data, so as to avoid over-learning or under-learning.

D. Model Evaluation

In this paper, well-known precision, recall, F1-measure, and AUC are selected to evaluate the prediction models' performance. Precision and recall are the indicators of evaluation models commonly used in machine learning. Precision represents the proportion of all defects whose severity is correctly classified as i(i is 0, 1, 2, 3, or 4) to all defects classified as i; recall refers to the proportion of all defects whose severity is correctly classified as i to all defects whose severity is i. To harmonize and average precision and recall, we also used the metric of F1-measure, which is commonly used to measure the accuracy of JIT defect prediction models [8], [10], [27]. In a project, the labeled data is often imbalanced, that is, defect-introducing changes with different severity don't distribute equally. Thus we added AUC into the evaluation metric suite. AUC is often used to evaluate the predictive ability of models built with unbalanced data [8], [23]. AUC is the Area Under the Curve of the receiver operating characteristic. Its scope is in [0,1]. The higher the value of AUC, the better the prediction performance of a model.

V. Evaluation

A. Subjects

We selected ten open-source projects from Apache [4] as our subjects. The number of commits for each project ranges from 2,414 to 16,025 before May 6, 2022. Each severity level of the projects we selected contains at least 30 defect-introducing changes. We set this standard to guarantee the non-triviality of our selected subjects, and the dataset used for analyses is large enough to substantiate the reliability of the results.

Table II reports the statistics of defect-introducing changes at different severity levels: #1 (i.e. Minor or Trivial), #2 (i.e. Major), #3 (i.e. Critical), and #4 (i.e. Blocker). The first row shows the number of changes that introduced defects at #1 severity level (i.e. Minor or Trivial), and the following line shows the proportion of it to all defect-introducing changes. Similarly, rows 3 - 8 present the number of changes at #2, #3, #4 severity level and their proportion, respectively. Row 9 represents the proportion of the sum of the changes at #2, #3, #4 severity levels to all defect-introducing changes. Row 10 shows the total number of commits. The last two rows show the total number of defect-introducing commits and the proportion of it to all studied commits.

[4]https://apache.org/

From this table, we can make a few straightforward observations: 1) a relatively large portion (from 19.5% to 46.4%, 30.0% on average) of all commits have induced defects, meaning that it is necessary to predict potential defects in advance when developers submit code changes; 2) The numbers of defect-introducing changes at different severity levels are various. The majority of defect-introducing changes are at the severity level #2, the percentages range from 29.1% to 78.5% (60.9% on average). The second largest proportion is at the severity level #4, reaching an average of 16.4%, and the third one is at the severity level #3, with an average of 13.0%. Defect-introducing changes at the severity level #1 occupy the smallest proportion with an average of 9.7%. According to the interpretation in prior work [28], *Blocker*, *Critical*, and *Major* represent high severity levels, while *Minor* and *Trivial* represent low severity. The above results show that the changes that introduce high-severity defects account for a really large proportion of all defect-introducing changes (from 80.8% to 97.9%, 90.3% on average). Because most of the high severity defects need to be fixed as soon as possible to avoid serious impact on the software system, it is necessary to predict the severity of the introduced defects just in time, which will help the development team to fix more serious defects in the shortest possible time.

B. Research Questions

RQ: What is the performance of our models on JIT defect severity prediction?

This question will show the predictive ability of our models, that is, whether the defect-introducing changes to be submitted by developers can be effectively predicted just in time. Since most serious defects need to be repaired as soon as possible and the quality of defect repair should be guaranteed to avoid serious impact on the software system, the timeliness of feedback defect severity should be fast and the precision of defect location should be high. At the same time, it is necessary to allocate defect repair personnel reasonably and avoid introducing new defects in the process of defect repair as far as possible. Therefore, it is very necessary to predict the severity of defects just in time, which can improve the efficiency and quality of defect repair and make greater use of limited resources.

C. Results

In each project, the amount of data with the defect severity level of 0 is far greater than that with the defect severity levels of 1, 2, 3, and 4. Therefore, this paper samples the data with the defect severity level of 0 in each project, so that its data amount is the same as the average data amount of the remaining levels. We used four metrics of precision, recall, F1-measure, and AUC to evaluate the predictive ability of our models.Tables III, IV, and V shows the experimental results when using Random Forest, Decision Tree, and KNN classifiers, respectively.

We can find that the average precision of the models built by Random Forest is 0.552, the average recall is 0.579, the

TABLE II
DEFECT DATA OF ALL PROJECTS

Project	Nutch	Avro	Zookeeper	Struts	Mahout	Openjpa	Iotdb	Pdfbox	Thrift	Airflow	Avg
#1	192	61	44	227	163	160	42	196	203	277	
#1/Defect-introducing Commits	19.2%	5.7%	3.9%	17.5%	12.0%	11.7%	2.1%	8.1%	8.0%	8.9%	9.7%
#2	419	565	326	732	996	966	1,572	1,906	1,612	2,035	
#2/Defect-introducing Commits	41.9%	53.2%	29.1%	56.5%	73.5%	70.7%	77.1%	78.5%	63.5%	65.2%	60.9%
#3	146	146	186	157	89	211	344	196	359	371	
#3/Defect-introducing Commits	14.6%	13.7%	16.6%	12.1%	6.6%	15.4%	16.9%	8.1%	14.1%	11.9%	13.0%
#4	244	291	564	180	107	30	80	129	364	436	
#4/Defect-introducing Commits	24.4%	27.4%	50.4%	13.9%	7.9%	2.2%	3.9%	5.3%	14.3%	14.0%	16.4%
(#2+#3+#4) /Defect-introducing Commits	80.8%	94.3%	96.1%	82.5%	88.0%	88.3%	97.9%	91.9%	92.0%	91.1%	90.3%
Defect-introducing Commits (#1+#2+#3+#4)	1,001	1,063	1,120	1,296	1,355	1,367	2,038	2,427	2,538	3,119	
Commits	3,283	3,223	2,414	6,233	4,506	5,251	6,360	10,800	6,543	16,025	
Defect-introducing Commits /Commits	30.5%	33.0%	46.4%	20.8%	30.1%	26.0%	32.0%	22.5%	38.8%	19.5%	30.0%

TABLE III
THE PERFORMANCE OF THE MODELS USING RANDOM FOREST

Project	Commits	Precision	Recall	F1-measure	AUC
Nutch	3,283	0.441	0.448	0.427	0.707
Avro	3,223	0.486	0.515	0.482	0.727
Zookeeper	2,414	0.457	0.516	0.475	0.747
Struts	6,233	0.539	0.552	0.509	0.740
Mahout	4,506	0.628	0.649	0.590	0.762
Openjpa	5,251	0.573	0.600	0.529	0.696
Iotdb	6,360	0.601	0.650	0.594	0.712
Pdfbox	10,800	0.704	0.700	0.650	0.771
Thrift	6,543	0.482	0.542	0.470	0.687
Airflow	16,025	0.604	0.613	0.553	0.738
Avg		0.552	0.579	0.528	0.729

TABLE IV
THE PERFORMANCE OF THE MODELS USING DECISION TREE

Project	Commits	Precision	Recall	F1-measure	AUC
Nutch	3,283	0.371	0.379	0.374	0.615
Avro	3,223	0.434	0.440	0.436	0.627
Zookeeper	2,414	0.425	0.428	0.426	0.624
Struts	6,233	0.454	0.459	0.455	0.619
Mahout	4,506	0.550	0.574	0.558	0.635
Openjpa	5,251	0.486	0.499	0.492	0.604
Iotdb	6,360	0.556	0.586	0.568	0.615
Pdfbox	10,800	0.592	0.617	0.601	0.635
Thrift	6,543	0.430	0.449	0.438	0.585
Airflow	16,025	0.489	0.506	0.497	0.627
Avg		0.479	0.494	0.485	0.619

TABLE V
THE PERFORMANCE OF THE MODELS USING KNN

Project	Commits	Precision	Recall	F1-measure	AUC
Nutch	3,283	0.346	0.346	0.346	0.572
Avro	3,223	0.415	0.413	0.414	0.594
Zookeeper	2,414	0.409	0.410	0.410	0.600
Struts	6,233	0.417	0.421	0.419	0.591
Mahout	4,506	0.528	0.532	0.530	0.607
Openjpa	5,251	0.495	0.496	0.495	0.580
Iotdb	6,360	0.530	0.535	0.532	0.571
Pdfbox	10,800	0.592	0.595	0.593	0.630
Thrift	6,543	0.416	0.422	0.419	0.559
Airflow	16,025	0.507	0.508	0.508	0.623
Avg		0.466	0.468	0.467	0.593

average F1-measure is 0.528, and the average AUC value is 0.729; For the models applying the Decision Tree algorithm, the average precision is 0.479, the average recall rate is 0.494, the average F1-measure is 0.485, and the average AUC value is 0.619. For the models applying the KNN algorithm, they have an average precision of 0.466, an average recall rate of 0.468, an average F1-measure of 0.467, and an average AUC value of 0.593.

The above results have demonstrated that our approach could effectively conduct JIT defect severity prediction. In addition, in terms of the development languages of the projects, because our data labeling work is based on the tool SZZ Unleashed (it is suitable for multiple development languages), our models are suitable for predicting projects in multiple development languages. From Tables III, IV, and V, we can also find that our models have a good predictive ability for projects with Java, Python, or C++ as the main development language. The number of commits of the projects studied in this paper ranges from 2,414 to 16,025, the span is relatively large. Whether it is small sample data or large sample data, our models still perform well, and we can also find that the more the number of project commits, the better the prediction ability of the models. Therefore, the prediction results of our models are credible, and the scope of application of the models is relatively wide.

VI. DISCUSSION

In this paper, we only studied ten open-source projects, thus we cannot guarantee that the experimental results are generalizable to all projects, and there may exist some extra discoveries that have not yet been exhibited in our paper. But we have partially addressed this problem by selecting projects with different sizes, ages, and domains.

We used the ten-fold cross-verification method for our models' training and validation. Although some studies [8], [18], [27] and our research have achieved good prediction results, Tan et al. [29] pointed out that the cross-validation method divides the data set randomly, which may take the future code change information as the training set and the past code change information as the test set. Therefore, in our future work, we plan to adopt some other methods, such as the time-wise-cross-validation method [9], out-of-sample bootstrap technique [11], or cross-project-validation method [9] in our approach.

VII. Conclusion

In this paper, we build the JIT defect severity prediction models by using Random Forest, Decision Tree, and KNN Classifiers, respectively. Through the evaluation of ten open-source projects, we have presented that our derived models can effectively predict defect severity just in time. More specifically, the prediction model using Random Forest can get a precision of 0.552 on average, and the average recall, F1-measure, and AUC are 0.579, 0.528, and 0.729, respectively. For the prediction models using Decision Tree, the average values of precision, recall, F1-measure, and AUC are 0.479, 0.494, 0.485, and 0.619, respectively. For the prediction models using KNN, the average precision, recall, F1-measure, and AUC are 0.466, 0.468, 0.467, and 0.593, respectively. In addition, we find the majority (90.3% on average) of defect-introducing changes are at a high severity level.

The rational allocation and efficient utilization of resources are particularly important in the process of defect repair. Thus it is necessary to predict the severity of potential defects when the code is committed. We believe that with the help of our prediction models, the development teams can make more effective use of limited resources and improve the quality and efficiency of defect repair.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under grant No. 62002129, The Knowledge Innovation Program of Wuhan-Shuguang Project under grant No. 2022010801020280.

References

[1] M. Iliev, B. Karasneh, M. R. Chaudron, and E. Essenius, “Automated prediction of defect severity based on codifying design knowledge using ontologies,” in *2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE)*. IEEE, 2012, pp. 7–11.

[2] Y. Tian, D. Lo, and C. Sun, “Information retrieval based nearest neighbor classification for fine-grained bug severity prediction,” in *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 215–224.

[3] Y. Tan, S. Xu, Z. Wang, T. Zhang, Z. Xu, and X. Luo, “Bug severity prediction using question-and-answer pairs from stack overflow,” *Journal of Systems and Software*, vol. 165, p. 110567, 2020.

[4] L. A. F. Gomes, R. da Silva Torres, and M. L. Côrtes, “Bug report severity level prediction in open source software: A survey and research opportunities,” *Information and software technology*, vol. 115, pp. 58–78, 2019.

[5] J. Arokiam and J. S. Bradbury, “Automatically predicting bug severity early in the development process,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, 2020, pp. 17–20.

[6] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, “Comparing mining algorithms for predicting the severity of a reported bug,” in *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, 2011, pp. 249–258.

[7] S. E. Sahin and A. Tosun, “A conceptual replication on predicting the severity of software vulnerabilities,” in *Proceedings of the Evaluation and Assessment on Software Engineering*, 2019, pp. 244–250.

[8] 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.

[9] X. Yang, H. Yu, G. Fan, and K. Yang, “Dejit: a differential evolution algorithm for effort-aware just-in-time software defect prediction,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 31, no. 03, pp. 289–310, 2021.

[10] X. Yang, D. Lo, X. Xia, and J. Sun, “Tlel: A two-layer ensemble learning approach for just-in-time defect prediction,” *Information and Software Technology*, vol. 87, pp. 206–220, 2017.

[11] R. Duan, H. Xu, Y. Fan, and M. Yan, “The impact of duplicate changes on just-in-time defect prediction,” *IEEE Transactions on Reliability*, 2021.

[12] D. Cubranic and G. C. Murphy, “Hipikat: Recommending pertinent software development artifacts,” in *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 408–418.

[13] J. Śliwerski, T. Zimmermann, and A. Zeller, “When do changes induce fixes?” *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

[14] S. Kim, T. Zimmermann, K. Pan, E. James Jr *et al.*, “Automatic identification of bug-introducing changes,” in *21st IEEE/ACM international conference on automated software engineering (ASE’06)*. IEEE, 2006, pp. 81–90.

[15] M. Borg, O. Svensson, K. Berg, and D. Hansson, “Szz unleashed: an open implementation of the szz algorithm-featuring example usage in a study of just-in-time bug prediction for the jenkins project,” in *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, 2019, pp. 7–12.

[16] 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 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 157–168.

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

[18] W. Fu and T. Menzies, “Revisiting unsupervised learning for defect prediction,” in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 72–83.

[19] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 284–292.

[20] M. D’Ambros, M. Lanza, and R. Robbes, “An extensive comparison of bug prediction approaches,” in *2010 7th IEEE working conference on mining software repositories (MSR 2010)*. IEEE, 2010, pp. 31–41.

[21] A. Mockus and D. M. Weiss, “Predicting risk of software changes,” *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[22] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura, “An analysis of developer metrics for fault prediction,” in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, 2010, pp. 1–9.

[23] S. McIntosh and Y. Kamei, “Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction,” *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 412–428, 2017.

[24] R. Ferenc, Z. Tóth, G. Ladányi, I. Siket, and T. Gyimóthy, “A public unified bug dataset for java and its assessment regarding metrics and bug prediction,” *Software Quality Journal*, vol. 28, no. 4, pp. 1447–1506, 2020.

[25] H. Osman, M. Ghafari, and O. Nierstrasz, “Hyperparameter optimization to improve bug prediction accuracy,” in *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE, 2017, pp. 33–38.

[26] L. Breiman, “Random forests,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[27] 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.

[28] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, “Predicting the severity of a reported bug,” in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010, pp. 1–10.

[29] M. Tan, L. Tan, S. Dara, and C. Mayeux, “Online defect prediction for imbalanced data,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 99–108.