ISBN: 978-981-94-1156-6
2024 the 14th International Workshop on Computer Science and Engineering (WCSE 2024)
doi: 10.18178/wcse.2024.06.045

Dataset Quality Improvement for Fine-Grained Just-in-Time Software Defect Prediction

Irem Fidandan [1] and Feza Buzluca [2 +]

[1] Infina Yazilim A.S., Turkey

[2] Department of Computer Engineering, Istanbul Technical University, Turkey

Abstract. Recently developed fine-grained JIT-SDP models separately predict whether a changed file in a commit will cause a defect in the future or not (in other words, defect-inducingness), in contrast to traditional JIT-SDP models that only predict commits. Fine-grained JIT-SDP models also cost-effectively reduce the risk of overlooking defect-inducing changes in effort-aware JIT-SDP models by allowing developers to review only defect-inducing changed files in a commit. But the fact is that building machine learning models is a data-dependent process, so the quality of the data is crucial. Low data quality negatively affects the predictive performance, interpretability, and scalability of machine learning models. In the context of JIT-SDP, there is no study in the literature that directly focuses on data quality. In this light of information, we proposed a novel data quality improvement method for fine-grained JIT-SDP models considering software domain. We then demonstrated that our data quality improvement method increases predictive performance for within-project and cross-project fine-grained JIT-SDP models. In doing so, we open the door to JIT-SDP models that have good predictive performance, cost-effectiveness, and a low probability of overlooking project components that cause defects.

Keywords: just-in-time software defect prediction, dataset quality, software metrics

1. Introduction and Related Work

Initial software defect prediction (SDP) models are built to identify defect-prone components such as modules, files, or methods. Although these types of models guide software teams to channel their efforts toward defect-prone components more, they are still impractical due to requiring full inspection of the defect-prone components. Consequently, just-in-time (aka change level) software defect prediction (JIT-SDP) models [1] are aroused. JIT-SDP models basically predict whether a commit (change) will cause a defect in the future (in other words, defect-inducingness) or not. JIT-SDP models are advantageous over conventional SDP models in the following ways: The provided fine granularity makes finding the root cause of the defects less time-consuming; developers are enabled to fix defect-inducing changes as soon as development is finished, and the context of the change is still fresh in their minds; and continuous code quality is ensured by preventing the merge of defect-inducing changes into the codebase.

To improve the cost-effectiveness of JIT-SDP models, effort-aware techniques that reduce the required effort for inspecting all defect-inducing changes without significantly sacrificing software quality by detecting only defect-inducing changes whose inspection requires less effort than the others are widely followed in the JIT-SDP literature [2]. However, because of the intrinsic characteristics of effort-aware JIT-SDP models, several defect-inducing changes would be overlooked. Recent fine-grained JIT-SDP models [3-6], on the other hand, predict defect-inducingness separately for each of the changed files in a commit. Thanks to these types of models, the risk of overlooking defect-inducing changes is reduced in a cost-effective way by enabling developers to inspect only defect-inducing files in a commit.

Building machine learning models is a data-dependent process, so the quality of the data is crucial. Low data quality negatively affects the predictive performance, interpretability, and scalability of machine learning models. Some of the characteristics that fundamentally define data quality and are familiar to many people who work with data are: uniqueness, validity, accuracy and relevance. Uniqueness is the absence of duplicate entities in the data set. Validity means that the values in the data set are in the correct range and

+ Corresponding author.
E-mail address: ifidandan@infina.com.tr, buzluca@itu.edu.tr.

format and do not conflict with other relevant values. In other words, the data does not contradict the realities of the domain. Accuracy means that the data set contains values that accurately represent the entities. Relevance is the absence of entities or values that are irrelevant to analyzed problem in the data set.

There isn't a study that specifically addresses these corresponding data quality characteristics in the JIT-SDP literature. We put forth a novel two-phased method to enhance these corresponding data quality characteristics, based on our observations and experience in software development. In the first phase, miscalculated features are sometimes deleted and sometimes corrected under the right conditions to ensure uniqueness, validity and accuracy. In the second phase, file changes in the commits that have little or no impact on future defects are excluded from the dataset to provide relevance. According to our experiments, we observed predictive performance improvements.

2. Proposed Dataset Quality Improvement Method Phases

2.1. Phase 1

The problems we observed and solved in the dataset are listed below:

- Some changed and renamed files in the commits appeared twice in the dataset. Feature values for one of these types of twice-duplicate file changes can be seen only in deleted files, while feature values for the other can only be seen in added files. These types of file changes result in incorrect calculation of historical and cumulative JIT metrics such as COMM, EXP, and AGE for changes after these problematic changes are made to the same files. We thus removed them from the dataset.
- Parent (pre-change) code features of all changed and renamed files in the commits are always incorrectly zero. For each subsequent commit that changed a given file, the current (post-change) code feature values of the former commit equal to parent (pre-change) code feature value of the latter commit in the real world. With this rule, we corrected this type of incorrect parent code feature values. Delta code feature values are derived by subtracting parent code feature values from current code feature values. Because of incorrect values of parent code feature values, delta code feature values are recalculated accordingly after the parent code feature values are corrected.

Thus, this phase ensures uniqueness, validity and accuracy data quality characteristics.

2.2. Phase 2

First, source code files written for testing or tutoring users how to use the components are irrelevant to defects. Secondly, the source codes in the deleted files will no longer be executed, making it impossible to cause a defect in the future. Therefore, we removed these type of file changes from the dataset. After that, we aimed to eliminate irrelevant file changes for future defects. For that purpose, we had set some rules, which are listed in Table 1. Table 2 contains an abbreviations list for the code features used in these rules.

Table 1: Irrelevant data elimination rules

Formal Definition	Description
If file f in commit c1 is deleted next commit c2 and (c2.time – c1.time) < gap, then file f in commit c1 is excluded.	When an added or modified file is deleted on the next commit that contains the same file, and the two commits have a time difference smaller than a predefined gap.
If file f in commit c is clean and c is the last that modified f and (project_last_commit.time – c.time) < gap, then file f in commit c is excluded.	When a file change has a clean label and the elapsed time after the change is less than a predefined gap.
If current_NOS = 0 for file f in commit c, then file f in commit c is excluded.	If after adding or modifying a source code file in a commit, the source code does not contain any instructions
If current_NOS > 0 and, current_NLA = 0 and current_NLM = 0 for file f in commit c, then file f in commit c is excluded.	If after adding or modifying a source code file in a commit, the source code does not contain any local (non-inherited) methods or attributes
If current_NLM = (current_NLG + current_NLS) for file f in commit c, then file f in commit c is excluded.	If after adding or modifying a source code file in a commit, the source code does not contain any local (non-inherited) methods other than local getter or local setter methods

la denotes added line count. If la = 0 for file f in commit c, then file f in commit c is excluded.	If no line is added after changing a source code file in a commit
If delta_LLOC = 0 and delta_WMC = 0 and delta_McCC = 0 and delta_RFC = 0 and delta_NLM = 0 and delta_NLA = 0 and delta_NOA = 0 and delta_NUMPAR = 0 and (la != 0 or ld != 0) for file f in commit c, then file f in commit c is excluded.	If only comment lines are changed in the source code file in a commit
If delta_NLM != 0 and (delta_NLM = delta_NLS + delta_NLG) and delta_NUMPAR != 0 and delta_NUMPAR = delta_NLS and delta_WMC != 0 and delta_WMC = (delta_NLS + delta_NLG) and delta_RFC != 0 and delta_RFC = (delta_NLS + delta_NLG) for modified file f in commit c, then file f in commit c1 is excluded.	If only local (non-inherited) getter or setter methods are added or deleted
If file.name ends with Exception, then file f is excluded.	If the source code file corresponds to an exception class
If file.name ends with Type, then file f is excluded.	If the source code file corresponds to an enum class
If current_NOA = 0 and, current_NII = 0 and current_CBOI = 0 for file f in commit c1 and after, then file f is excluded for commits c1 and after.	If after adding or modifying a source code file in a commit, the source code corresponds to the class whose methods are no longer called afterward (in short: dead code)
If current_WMC = current_NLM for file f in commit c, then file f in commit c is excluded.	If the source code file corresponds to a data class or a class whose methods only call methods of other classes and do nothing on their own
la denotes added line count. ld denotes deleted file count. If la = 0 and ld = 0 for file f in commit c, then file f in commit c is excluded.	If only the path of the source code file is changed, but not the content in a commit

Table 2: Abbreviations of code metrics

Abbreviation	Stands For
NOS	Number of Statements.
NLA	Number of Local Attributes
NLM	Number of Local Methods
NLG	Number of Local Getters
NLS	Number of Local Setters
LLOC	Logical Line of Code
WMC	Weighted Method Count
McCC	McCabe Cyclomatic Complexity
RFC	Response For Class.
NOA	Number of Ancestors
NUMPAR	Number of Parameters
NII	Number of Incoming Invocations.
CBOI	Coupling Between Objects Inverse

3. Experiments

We first applied our novel quality improvements to the Trautsch et al. [4] dataset. We then built fine-grained within-project and cross-project JIT-SDP models for projects in the dataset and evaluated their predictive performances. The following headings of these sections explain the steps we take in detail.

3.1. Dataset Construction

All projects in the dataset are cloned from GitHub to local for the purpose of tracking files and file operations across commits. However, WSS4J, ManifoldCF and Santuario were excluded from our analysis

because their commits in the dataset are not listed in their current GitHub history. In the course of a project, the names of files may change, or a new file may have the same name as a deleted or renamed file. Using PyDriller [7], we developed an automation that iterates through commits in git history in ascending date order and assigns each file a unique name to accurately identify it and determine the type of file operation. After this step, we handled miscalculations and eliminated irrelevant files in the commits as we proposed. Table 3 and Table 4 show dataset statistics before and after our data quality improvements for selected projects. Since space is limited and the number of projects is high, not all projects can be shown. However, we have shared these statistics across all projects on GitHub [8].

Table 3: Project-based dataset statistics before and after data quality improvement

Project	Before Data Quality Improvement					After Data Quality Improvement for Adhoc				After Quality Improvement for ITS			
	#row	#com	#file	%adh	%its	#row	#com	#file	%buggy	#row	#com	#file	%buggy
ant-ivy	11581	1917	716	0.3	0.05	4074	1304	334	0.46	4074	1304	334	0.1
archiva	23899	3873	2640	0.12	0.03	5391	2208	886	0.23	5398	2210	886	0.12
calcite	24653	2056	2159	0.3	0.05	8955	1650	1045	0.46	9008	1650	1055	0.09
cayenne	42203	4157	4927	0.1	0.03	9507	2471	1683	0.23	9514	2473	1684	0.1

Table 4: Project-based dataset change percentages after data quality improvement

Project	After Data Quality Improvement for Adhoc				After Quality Improvement for ITS			
	(-) % of row count	(-) % of commit count	(-) % of file count	(+) % of buggy label	(-) % of row count	(-) % of commit count	(-) % of file count	(+) % of buggy label
ant-ivy	64.82	31.98	53.35	53.33	64.82	31.98	53.35	100.0
archiva	77.44	42.99	66.44	91.67	77.41	42.94	66.44	300.0
calcite	63.68	19.75	51.6	53.33	63.46	19.75	51.13	80.0
cayenne	77.47	40.56	65.84	130.0	77.46	40.51	65.82	233.33

3.2. Model Construction and Evaluation

There are two label types in the dataset: Ad-hoc and ITS, which are based on SZZ [9] algorithm. Their distinctions are at the heart of identifying bug-fix purpose commits. Ad-hoc looks for specific keywords in commit messages that indicate bug-fix intent, such as bug or fix, whereas ITS checks if the commit is connected with a bug report in issue tracking systems.

In within-project fine-grained JIT-SDP model construction and evaluation steps, they [4] followed a time-sensitive interval approach. Disjoint commit sets were initially created for four-month intervals according to the commit time order. Then, for every commit set, the commits from the first three months are reserved for training, and the commits from the last month are reserved for testing. Therefore, separate models are created and evaluated. Additionally, to handle imbalance and reduce the potential risk for poor predictive performance, they used SMOTE [10] algorithm on train groups to generate minority instances synthetically for balancing the occurrence counts for both labels. Finally, to evaluate model performance in general, the predictive performance scores are averaged arithmetically. They performed these model construction steps for JIT metrics, static code metrics, and static PMD analyzer metrics, as well as Ad-Hoc and ITS labels. We followed the same approach with our improved dataset.

Although they also performed cross-validation, we exclude it from our scope because cross-validation is not a realistic evaluation due to the nature of the JIT-SDP problem, leading to deceptively higher prediction performance results, which is explained in the study conducted by Tan et al. [11].

Furthermore, we explored fine-grained, cross-project JIT-SDP models by using one project for training and another project for testing for all project combinations. Likewise, we used SMOTE [10] to balance labels and reconstructed models for each combination of metrics and label types, as we did in within-project settings.

Finally, we also used CFS (Correlation Based Feature Selection) [12] for both within-project and cross-project settings to investigate whether we can create cleaner and more explainable models without sacrificing predictive performance and perhaps potentially achieving better predictive performance sometimes.

3.3. Results

In the within-project settings for all metric and label types, Table 5 displays the median f1 values of all projects: without dataset quality improvement (baseline), with dataset quality improvement but without CFS, and with dataset quality improvement plus CFS. Similar information is provided in Table 6, although it is specific to cross-project settings. Our quality improvements to the dataset always result in higher f1 values than the baseline, both within and cross project settings. Furthermore, CFS increases f1 scores all the time in cross-project settings.

Table 5: Median f1 scores for Within-Project settings

metric_set	label	f1 baseline	f1 without cfs	f1 with cfs
jit	adhoc	0.30	0.36	0.33
jit	its	0.06	0.15	0.18
jit_static_pmd	adhoc	0.32	0.35	0.32
jit_static_pmd	its	0.13	0.19	0.17
pmd	adhoc	0.24	0.32	0.31
pmd	its	0.10	0.18	0.18
static	adhoc	0.35	0.35	0.33
static	its	0.13	0.22	0.17

Table 6: Median f1 scores for Cross-Project settings

metric_set	label	f1 baseline	f1 without cfs	f1 with cfs
jit	adhoc	0.29	0.37	0.35
jit	its	0.05	0.09	0.13
jit_static_pmd	adhoc	0.26	0.35	0.37
jit_static_pmd	its	0.00	0.02	0.13
pmd	adhoc	0.20	0.32	0.25
pmd	its	0.04	0.09	0.13
static	adhoc	0.21	0.32	0.37
static	its	0.01	0.03	0.12

4. Threats to Validity

Aside from the factors of threats to validity mentioned in Trautsch et al.'s work, which are about identifying defect-inducing commits and selected projects, we also discuss two additional factors of threats to validity. The rules that we set for eliminating irrelevant files in a commit are based on object-oriented metrics and Java programming language naming conventions. So, the first factor that threatens the validity is that our method may not be suitable for other projects in a different programming language. First two rules exclude the files in the commits based on the parametric gap value corresponding to the maximum waiting time to be sure whether the change in the file causes a defect. Gap values other than ours may result in different eliminated files in the commits, so the results may differ from ours based on the data set. Additionally, appropriate gap values for projects may vary. So, the second factor that threatens validity is the parametric gap value.

5. Conclusion

No research has been conducted in JIT-SDP literature that specifically focuses on data quality considering the realities of the software domain. In this study, we proposed a novel data quality improvement method that ensures uniqueness, validity, accuracy and relevance for fine-grained JIT-SDP models. Fine-grained JIT-SDP models improve cost efficiency while minimizing the risk of overlooking defect-inducing changes. We constructed fine-grained JIT-SDP models for both within-project and cross-project settings,

comparing with and without our data quality improvements. With our proposed data quality improvements, we observed an increase in prediction performance in our experiments. As a result, the study's encouraging findings pave the way for the deployment of software defect prediction models with high predictive performance, low cost, and a low risk of missing defect-causing project components. Ultimately, our proposed data quality improvement was tried on a certain number of projects. To generalize, it would make sense to try it out on other projects.

6. 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. Institute of Electrical and Electronics Engineers (IEEE), pp. 181–196, Mar. 2008. doi: 10.1109/tse.2007.70773.

[2] Y. Kamei et al., "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6. Institute of Electrical and Electronics Engineers (IEEE), pp. 757–773, Jun. 2013. doi: 10.1109/tse.2012.70.

[3] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," Journal of Systems and Software, vol. 150. Elsevier BV, pp. 22–36, Apr. 2019. doi: 10.1016/j.jss.2018.12.001.

[4] A. Trautsch, S. Herbold, and J. Grabowski, "Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction," *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE*, Sep. 2020. doi: 10.1109/icsme46990.2020.00022.

[5] S. Amasaki, H. Aman, and T. Yokogawa, "An Evaluation of Effort-Aware Fine-Grained Just-in-Time Defect Prediction Methods," *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE*, Aug. 2022. doi: 10.1109/seaa56994.2022.00040.

[6] F. Lomio, L. Pascarella, F. Palomba, and V. Lenarduzzi, "Regularity or Anomaly? On The Use of Anomaly Detection for Fine-Grained JIT Defect Prediction," *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA).* IEEE, Aug. 2022. doi: 10.1109/seaa56994.2022.00049.

[7] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, Oct. 26, 2018. doi: 10.1145/3236024.3264598.

[8] https://github.com/iremfidandan/fineGrainedJitDatasetImprovementResults

[9] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4. Association for Computing Machinery (ACM), pp. 1–5, May 17, 2005. doi: 10.1145/1082983.1083147.

[10] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic Minority Over-sampling Technique," *Journal of Artificial Intelligence Research*, vol. 16. AI Access Foundation, pp. 321–357, Jun. 01, 2002. doi: 10.1613/jair.953.

[11] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, May 2015, doi: 10.1109/icse.2015.139.

[12] M. A. Hall, Correlation-based feature selection for machine learning. PhD thesis, The University of Waikato, 1999.