ClusterCommit: A Just-in-Time Defect Prediction Approach Using Clusters of Projects

Mohammed A. Shehab
ECE, Concordia University
Montreal, QC, Canada
mohammed.shehab@concordia.ca

Abdelwahab Hamou-Lhadj
ECE, Concordia University
Montreal, QC, Canada
wahab.hamou-lhadj@concordia.ca

Luay Alawneh
Jordan University of Science and Technology
Irbid, Jordan
lmalawneh@just.edu.jo

***Abstract*—Existing Just-in-Time (JIT) bug prediction techniques are designed to work on single projects. In this paper, we present ClusterCommit, a JIT bug prediction approach geared towards clusters of projects that share common libraries and functionalities. Unlike existing techniques, ClusterCommit trains a machine learning model by combining commits from a set of projects that are part of a larger cluster. Once this model is built, ClusterCommit can be used to detect buggy commits in each of these projects. When applying ClusterCommits to 16 projects that revolve around the Hadoop ecosystem and 10 projects of the Hive ecosystem, the results show that ClusterCommit achieves an F1-score of 73% and MCC of 0.44 for both clusters. These preliminary results are very promising and may lead to new JIT bug prediction techniques geared towards projects that are part of a large cluster.**

***Index Terms*—Bug Prediction, Commit Analysis, Machine Learning, Just-In-Time Software Maintenance, Software Reliability.**

I. Introduction

The analysis of code commits can help catch unwanted modifications to the system before these modifications are integrated with the final release, providing an additional line of defence against the introduction of bugs [1]. There exist a number of studies to predict bugs at commit time. These Just-in-Time (JIT) bug prediction methods can not only detect bugs early in the coding process, but also provide immediate feedback to developers to check and correct their code while the changes are still fresh in their mind [1] [2] [3]. This is contrasted with traditional bug prediction techniques (e.g. [4]) that operate on the entire source code, delaying the provision of feedback.

JIT bug prediction methods rely on machine learning to classify incoming commits as risky (commits that may potentially introduce faults in the system) or not [1] [5] [2]. Existing approaches train and test machine learning models on commits extracted from a single project (e.g., [5]). These techniques vary mainly in terms of the features and the algorithms they use. Other researchers (e.g., [3]) have proposed a transfer learning method in which they train models using data from one project and evaluate the models on a testing dataset from a different project [3]. Although the proposed approaches vary in their design, they rely on a training model that is built using historical commits from a single project only.

Recently, Nayrolles and Hamou-Lhadj [2] conducted a study at Ubisoft, the video game development company, in which they developed CLEVER, a novel JIT bug prediction approach. CLEVER is unique in the sense that it relies on training a JIT bug prediction model that combines commits from many Ubisoft video game projects that run on the same game engine. The authors argued that, for industrial projects, it is useful to combine commits from highly-coupled projects instead of working on each project separately. This is because these projects reuse libraries and share an important code base, rendering them vulnerable to the same faults.

Inspired by the design of CLEVER, we conducted a study to investigate the use of clusters of projects for JIT bug prediction in open source systems. As a motivating example, take the Apache Foundation projects[1] (used in the evaluation section). These projects offer complementary services in diverse fields. Many of them are built by reusing other projects and libraries. For example, the Bigtop, ZooKeeper and Spark projects are built on Hadoop. Similiarly, Ambari, Falcon, and Oozie projects use Hive, another Apache project. By examining the bug reports of many of these projects, we found bug reports of one project that refer to bugs in another project. For example, the description of bug report OOZIE-3563 of the Oozie project refers to the inability to initiate the Hive project. For these interrelated projects, we conjecture that it would be beneficial to treat them as one cluster and build a training model that combines their commits. This led us to the design of ClusterCommit, a JIT bug prediction approach based on clusters of projects.

Although ClusterCommit is inspired by CLEVER, the two approaches exhibit important differences in the way they are designed. A key design feature of CLEVER is its reliance on clone detection techniques to predict buggy commits. For each suspected commit, CLEVER extracts the corresponding code block and compares it to a database of known defects. This design choice suggests that CLEVER is too dependant on the way Ubisoft projects are developed where large code segments may be reused across systems (perhaps because they are written by the same development teams). ClusterCommit, on the other hand, relies solely on code and process metrics, common to any software system (see Section II.B). This way, we do not assume anything about the way the projects are developed. This is particularly important for

[1] https://projects.apache.org/projects.html?category

open systems since they are developed by contributors from different organizations. Additionally, ClusterCommit relies on a clustering technique to identify projects that should be grouped together since we cannot rely on domain knowledge to identify groups of interrelated projects as it is the case for industrial systems. Finally, ClusterCommit uses a time-validation approach (see Section II-E), which accounts for the temporal order of commits, to evaluate the prediction accuracy. This temporal order is not considered in CLEVER, yielding a situation where past commits may be potentially compared to future commits.

II. The ClusterCommit Approach

Figure 1 depicts the overall steps of ClusterCommit. ClusterCommit requires as input a set of projects that share some dependencies. This input can be specified by the user of ClusterCommit. The next step of ClusterCommit is to cluster the projects so as to identify strongly-coupled subprojects, which are likely to share a large code base. For each cluster, we build a training model by aggregating past commits (both healthy and buggy) extracted from each project of the cluster. The resulting model is used for testing. At which point, ClusterCommit predicts whether a commit is buggy or not for each project individually.

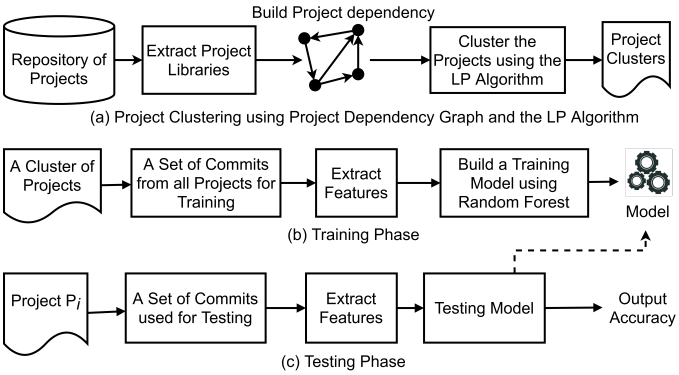

Fig. 1. Overall approach

A. Project Clustering

Consider the set $P = \{P_1, P_2, ...P_N\}$ of size N, a set of projects that are given as input by the user. ClusterCommit starts by extracting the libraries that each project uses. This information is found in the dependency management system used by the projects. For example, Java projects (the focus of this paper) are managed and built using tools such as the Maven[2] dependency manager to save development time by reusing internal and external (third-party) libraries [6]. We mine Maven information to extract libraries used by each project. Assume $L = \{L_1, L_2, ..., L_M\}$ is a set of distinct libraries used by the projects of P, where M is the number of distinct libraries. ClusterCommit builds a project dependency graph $G = (V, E)$ where V is the set of nodes representing projects of P and libraries of L, and E represents the set of edges between projects and libraries. We define a directed edge from a project P_i to a library L_j if P_i uses the library L_j. This type of graphs is known as a community graph [7]. Instead of having edges between projects, we link projects to their libraries. The idea is to find projects that share a large number of libraries and cluster them together. For this, we use a community-based clustering technique. More particularly, we choose to apply the Label Propagation (LP) algorithm for finding communities [7]. For more details about the LP algorithm, we refer the reader to the work of Raghavan et al. [7]. Other clustering techniques can also be used such as those used for various software engineering purposes (e.g., [8]).

B. Feature Extraction

To train our machine learning model using commit information, we use the same features as the ones proposed by Kamei et al. [1]. The authors proposed 14 features extracted from commit data and bug reports to classify commits (see Table I). These features are grouped into five categories: diffusion, size, purpose, history, and experience. It is common when using classification algorithms to have highly correlated features [9]. We used the Pearson correlation coefficient [10] to measure the correlation among the features. We found that the feature SEXP (Developer Experience on Sub-systems) is more than 70% correlated to EXP and REXP. We chose 70% as the threshold because this threshold has been widely used in defect prediction research ([9] [11] [12] [13]). Thus, we dropped SEXP from our features set. The final feature set consists of 13 features, shown in Table I, excluding SEXP.

TABLE I
The features used to build the prediction model

Dimension	Name	Description
Diffusion	NS	Number of modified sub-systems
	ND	Number of modified directories
	NF	Number of modified files
	Entropy	Distribution of modified code across files
Size	LA	Added lines
	LD	Deleted lines
	LT	Line of code before edit
Purpose	Fix	Whether or not the change is a defect or fix
History	NDEV	Number of developers that changed the file
	AGE	The average time between file changes
	NUC	The number of unique changes
Experience	EXP	Developer experience
	REXP	Recent developer experience
	SEXP	Developer experience on sub-systems

C. Data Labeling

Since we are using supervised machine learning techniques, we need to label the commits associated with the subject projects as either healthy or buggy. To this end, we use an enhanced version of the SZZ algorithm [14] known as the *Refactoring Aware SZZ Implementation*[4] (RA-SZZ) algorithm proposed by Campos Neto et al. [15]. SZZ works by examining the bug reports from the bug tracking system (Jira in

[2]https://mvnrepository.com/repos/central

[4]https://github.com/danielcalencar/raszzprime

our case). The algorithm goes through all bug reports that are resolved with the attempt to link them to commits using the bug report unique identifier inside the commit message (if available). Finally, the algorithm proceeds by going back in the commit history to obtain the original commits that had introduced the discovered bug and labels them as buggy. At the end, the algorithm outputs the list of buggy commits.

D. Classifier

In this paper, we use Random Forest (RF) [16] as the classification algorithm. We chose RF because Pascarella et al. [17] tested seven supervised machine learning models over 10 projects and found that the best performance was obtained with RF. We intend to conduct future studies to compare the impact of various algorithms on the results.

E. Evaluating the classifier

One way to validate the performance of a classifier would be to use the traditional 10-fold cross validation approach. The problem with this approach is that it does not consider the temporal order of the commits, leading to a situation where commits from the past may be tested against a model that is trained on commits from the future. To address this issue, Tan et al. [5] proposed a time-based validation approach. This approach sorts the commits based on their timestamps and groups them into slots depending on the month of submission. Then, it chooses a time interval for training and another one for testing. In addition, Tan et al. [5] argued that there should be a gap between the commits used for training and those used for testing to account for the time between submitting a buggy commit and the manifestation and reporting of the bug. Therefore, the time-validation approach requires the definition of three time intervals: train, gap, and test.

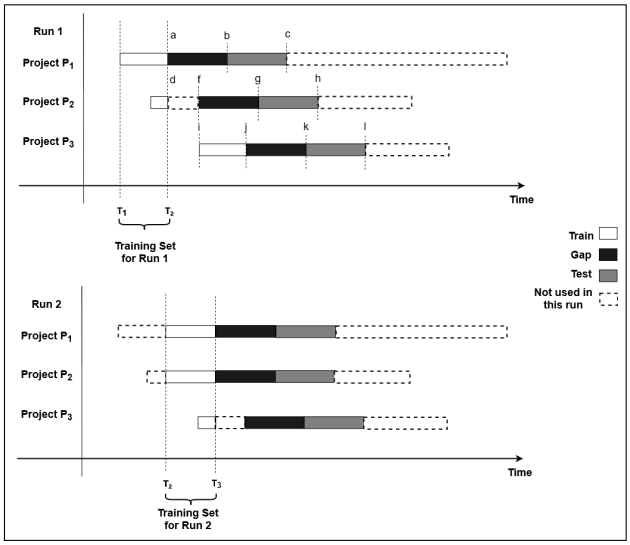

Fig. 2. An example of time-validation using ClusterCommit with three projects and two runs

Figure 2 illustrates how we used time validation with clusters of projects using as example three fictive projects and two runs of validation. To explain how our approach works, we first need to determine the length of the training, gap, and testing time intervals. Based on the literature, we consider 6 months as the length of the training time interval as suggested by McIntosh et al. [18] since the projects we use in the evaluation have several years of historical commits (see Section III-A). For the testing and gap time intervals, we take the minimum of the average fixing times of all projects of the cluster. More formally, consider aft_i as the average time it takes to fix bugs of Project P_i. The length of the gap and testing time intervals is computed as follows: $length = min(aft_1, aft_2, ..., aft_N)$, with N being the total number of projects in a cluster. The rationale behind taking the minimum of the averages instead of, for example, the average of averages, is to ensure that for each project we can perform at least one run of validation. The length of the training, gap, and testing time intervals is fixed for all projects.

Now that we have determined the length of the three time intervals needed for time-validation, we start by iterating through the data to determine the commits used for training and testing in each iteration. In each run, we build a training model that combines commits that appear in the training time interval. We test each project individually against the model using commits that appear in the testing time interval. For example, for the projects of Figure 2, in Run 1, we use the commits of Project P1 and some commits of Project P2 that fall within the training time interval as a training set. To test commits of P1, we use commits that appear in time interval b to c. For P2, we use commits in the time interval g to h, and finally we test P3 with commits in time interval k to l. We are aware that P3 is tested against commits of P1 and P2 and that none of P3 commits are used for training in this iteration. This is perfectly aligned with the core idea of ClusterCommit where strongly interrelated projects can have commits of one project or more used to predict buggy commits in other projects. In Run 2, the process continues by shifting the time window with exactly the length of the training time interval and the process is repeated again. With this approach, we have as many runs as necessary until all projects are covered. For each project, we measure the prediction accuracy (see next section for the evaluation metrics) in each run, and take the average as the final performance of the classifier for each project. Note that the number of runs through the projects is not the same. For example, in the last run of our example, both Projects P2 and P3 in Figure 2 will end up contributing commits to a training set that are used to test commits of Project P1 only. This is because the ending time of these projects is earlier than that of project P1.

F. Evaluation Metrics

We use precision, recall, and F1-Score to evaluate the effectiveness of ClusterCommit. These metrics are widely used in related studies (e.g., [3] [2]). They rely on the true positive (TP), false positive (FP), and false negative (FN) values and are computed as follows: Precision = TP/(TP + FP), Recall = TP/(TP + FN), and F1-Score = 2*Precision*Recall/(Precision

+ Recall). We also use the Matthews Correlation Coefficient (MCC) [19], which is computed as follows:

$$MCC = \frac{(TP * TN) - (FP * FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \quad (1)$$

III. Case Study

A. Subject Systems

To evaluate the effectiveness of ClusterCommit, we need a group of related projects. For this, we turn to projects of the Apache Foundation. The Apache Foundation maintains a list of projects categorized depending on their domain such as Big Data, Cloud, Build Management, Logging, etc. In this paper, we choose to focus on projects of the Big Data and Database domain. Another category can also be used. The Big Data and database category contains 72 open source projects (at the time of conducting this study) for managing and processing large data. We further restrict the number of projects to those written in Java and built using the Maven dependency manager. This is because we use Maven to extract shared libraries to build the project dependency graph used for clustering. In addition, we only select projects that have at least 1,000 commits to ensure that we have mature projects with sufficient history. By applying these criteria, we ended up with 34 projects. These projects are available on Github and use Jira[2] for bug tracking.

B. Result of Clustering

In this step, we apply the LP clustering algorithm to the 34 projects. We wrote a script to extract the Maven dependencies and built a project dependency graph that shows the relationships between projects based on the number of libraries they share. We found that these projects have 1,520 distinct libraries. The LP algorithm returned seven distinct clusters, among which two clusters are the most predominant. The first cluster has Hadoop as its super-node and contains 16 projects and 868 (57.10%) shared libraries. The second cluster consists of projects that revolve around Hive as a super-node. It contains 10 projects that share 652 (42.9%) libraries. We refer to these clusters as Hadoop and Hive clusters and use them in this paper to illustrate the performance of ClusterCommit.

Table II shows the projects that belong to these two clusters. As we can see the Hadoop cluster contains projects that are built around the Hadoop ecosystem such as Camel, a Hadoop-based project that integrates various systems that consume and produce data. Camel accepts data stored in Hadoop Distributed File Management System (HDFS). The same goes with other projects such as Drill, Helix, Spark, and Parquet, which revolve around Hadoop technology. The Hive cluster groups projects that support the management of data warehouses. Table II shows information about the clusters and their projects including the project name, version, total number of commits, the ratio of buggy commits (shown as linked commits using RA-SZZ) and the project time period.

[2]https://www.atlassian.com/software/jira/features/bug-tracking

TABLE II
Subject projects considered in this study

	Name	Version	Commits	Defects (%)	Project Period
Hadoop	Bigtop	1.5.0	2,599	1.2%	Aug/2011 - Dec/2020
	Bookkeeper	4.12.1	2,374	3.5%	Mar/2011 - Dec/2020
	Camel	2.10.7	13,023	30.6%	Mar/2007 - Sep/2013
	Curator	2.0.0	2,718	1.0%	Jul/2011 - Jan/2021
	Drill	1.10.0	2,597	63.3%	Oct/2012 - Feb/2021
	Flink	1.12.1	24,983	18.5%	Dec/2010 - Jan/2021
	Gora	0.1-incub.	1,367	3.8%	Oct/2010 - Nov/2020
	Hadoop	2.6.0	10,508	6.0%	Sep/2009 - Aug/2014
	Helix	1.0.1	3,756	1.5%	Jun/2011 - Jun/2020
	Ignite	1.0.1	10,836	14.8%	Feb/2014 - Sep/2017
	Oodt	1.9	2,084	4.1%	May/2010 - Jan/2021
	Parquet	1.8.0	1,679	6.8%	Aug/2012 - Feb/2021
	Reef	0.16.0	3,749	1.6%	Aug/2012 - Nov/2020
	Spark	2.2.1	19,967	1.8%	Apr/2010 - Nov/2017
	Tez	0.9.2	2,658	8.7%	Mar/2013 - Mar/2019
	Zookeeper	3.6.0	2,030	28.4%	Nov/2007 - Nov/2019
Hive	Accumulo	2.0.1	10,094	5.5%	Oct/2011 - Dec/2021
	Airavata	0.17	7,227	6.9%	Jul/2011 - Mar/2019
	Ambari	2.7.5	24,578	0.4%	Sep/2011 - Jun/2020
	Carbondata	2.1.0	4,746	11.6%	Mar/2016 - Feb/2021
	Falcon	0.11	2,209	5.9%	Nov/2011 - Aug/2018
	Flume	1.9.0	1,812	42.4%	Aug/2011 - May/2020
	Hive	3.1.3	12,277	4.2%	Sep/2008 - Jan/2020
	Oozie	5.2.0	2,332	4.9%	Sep/2011 - Jan/2021
	Storm	2.2.0	10,326	2.3%	Sep/2011 - Feb/2021
	Zeppelin	0.8.2	3,896	13.9%	Jun/2013 - Feb/2021

C. ClusterCommit Results and Discussion

We set the length of the training time interval to 6 months as discussed in Section II-E. To compute the length of the testing time interval, which is also the length of the gap time interval, we measure the average bug fixing time of each project in a cluster and take the minimum of averages as explained in Section II-E. We found that the minimum for both clusters is 8 months.

TABLE III
ClusterCommit Results

	Project name	Precision (%)	Recall (%)	F1 score (%)	MCC (%)
Hadoop	Bigtop	72.63	51.40	60.19	0.34
	Bookkeeper	66.22	85.76	74.74	0.42
	Camel	75.03	76.69	75.85	0.51
	Curator	71.87	90.21	80.00	0.53
	Drill	72.47	79.54	75.84	0.49
	Flink	70.78	73.18	71.96	0.42
	Gora	64.67	66.06	65.36	0.30
	Hadoop	64.88	81.54	72.26	0.38
	Helix	70.50	80.45	75.15	0.45
	Ignite	69.98	59.53	64.33	0.34
	Oodt	60.63	62.90	61.74	0.19
	Parquet	76.37	75.11	75.74	0.51
	Reef	79.76	93.70	86.17	0.70
	Spark	74.00	75.16	74.58	0.49
	Tez	67.95	82.69	74.60	0.44
	Zookeeper	75.64	67.41	71.28	0.46
	Average	***70.84***	***75.08***	***72.89***	***0.44***
Hive	Accumulo	66.64	61.05	63.72	0.31
	Airavata	66.64	72.96	69.65	0.37
	Ambari	71.43	81.49	76.13	0.49
	Carbondata	71.07	81.82	76.07	0.49
	Falcon	65.17	81.90	72.58	0.40
	Flume	78.99	79.57	79.28	0.56
	Hive	69.82	67.12	68.44	0.38
	Oozie	71.56	84.04	77.30	0.53
	Storm	63.97	74.17	68.69	0.33
	Zeppelin	78.58	73.69	76.05	0.54
	Average	***70.39***	***75.78***	***72.98***	***0.44***

Table III shows the results of ClusterCommit. For both clusters. Our approach achieves an F1-Score of around 73% and an MCC of 0.44. Note that MCC varies from -1 to 1 with the latter being the perfect model. We also observe that ClusterCommit has a higher recall than precision. In other words, while it detects more buggy commits (high recall), in both clusters it has around 29% false positives (precision of about 71%). This contradicts the results obtained by Nayrolles and Hamou-Lhadj for their approach CLEVER (79% precision and 65% recall). Further studies with more clusters are needed to generalize the results of this study.

D. Threats to Validity

We experiments with 26 projects of the Apache Foundation. We need to conduct more experiments to generalize our results. The size of the time intervals can affect the performance of the prediction models. We need to experiment with different time intervals to see their impact on the result. Furthermore, we relied on the RA-SZZ algorithm and the implementation provided by the authors to label the data. Errors in this implementation may impact our results.

IV. Related Work

The closest work to ours is the study that Nayrolles and Hamou-Lhadj [2] conducted at Ubisoft, which we discussed in the introductory section. Other JIT bug prediction studies train models on single projects. Some notable studies include the work of Fukushima et al. [20]. The authors examined the performance of JIT defect prediction models and showed that cross-projects techniques provide superior performance when compared to single-project methods. Huang et al. [21] proposed an improved supervised JIT model called CBS+ (Classify-Before-Sort)+ based on the logistic regression model. Catolino et al. [3] investigated single and ensemble supervised machine learning models to build the JIT models with mobile apps. Kamei et al. [1] proposed a single-project bug prediction model using 14 features extracted from code and bug report repositories. McIntosh et al. [18] examined the performance of linear regression models for JIT defect prediction using short-term and long-term data. Yan et al. [22] designed a framework to detect the risky changes for code and then recognize buggy code location.

V. Conclusion

We proposed ClusterCommit a new JIT bug prediction approach that builds a training model that combines commits from projects of the same cluster and tests commits from each project individually. When applied to two clusters of projects of the Apache Foundation with a total of 26 projects, we showed that ClusterCommit yields promising results. We intend to (a) experiment with more clusters, (b) apply transfer learning across clusters of projects, and (c) investigate types of bugs that can be detected with this approach.

References

[1] 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, 2013.

[2] M. Nayrolles and A. Hamou-Lhadj, "Clever: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects," in *15th International Conference on Mining Software Repositories (MSR'18)*, 2018, pp. 153–164.

[3] G. Catolino, D. Di Nucci, and F. Ferrucci, "Cross-project just-in-time bug prediction for mobile apps: An empirical assessment," in *Proceedings of the 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2019, pp. 99–110.

[4] T. Zhou, X. Sun, X. Xia, B. Li, and X. Chen, "Improving defect prediction with deep forest," *Information and Software Technology*, vol. 114, pp. 204 – 216, 2019.

[5] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *Proceedings of the 37th International Conference on Software Engineering Volume2*, ser. ICSE '15. IEEE Press, 2015, pp. 99–108.

[6] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, p. 384–417, Feb. 2018.

[7] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical review E*, vol. 76, no. 3, p. 036106, 2007.

[8] C. Patel, A. Hamou-Lhadj, and J. Rilling, "Software clustering using dynamic analysis and static dependencies," in *13th European Conference on Software Maintenance and Reengineering*, 2009, pp. 27–36.

[9] J. Jiarpakdee, C. Tantithamthavorn, H. K. Dam, and J. Grundy, "An empirical study of model-agnostic techniques for defect prediction models," *IEEE Transactions on Software Engineering*, 2020.

[10] W. Kirch, Ed., *Pearson's Correlation Coefficient*. Dordrecht: Springer Netherlands, 2008, pp. 1090–1091.

[11] J. Jiarpakdee, C. Tantithamthavorn, and C. Treude, "The impact of automated feature selection techniques on the interpretation of defect models," *Empirical Software Eng.*, vol. 25, no. 5, pp. 3590–3638, 2020.

[12] J. Jiarpakdee, C. Tantithamthavorn, and A. E. Hassan, "The impact of correlated metrics on the interpretation of defect models," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 320–331, 2021.

[13] J. Jiarpakdee, C. Tantithamthavorn, and C. Treude, "Autospearman: Automatically mitigating correlated software metrics for interpreting defect models," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 92–103.

[14] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, May 2005.

[15] E. C. Neto, D. A. da Costa, and U. Kulesza, "The impact of refactoring changes on the szz algorithm: An empirical study," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 380–390.

[16] Tin Kam Ho, "Random decision forests," in *Proceedings of 3rd International Conference on Document Analysis and Recognition*, vol. 1, 1995, pp. 278–282.

[17] 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.

[18] 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, 2018.

[19] P. Baldi, S. Brunak, Y. Chauvin, C. A. F. Andersen, and H. Nielsen, "Assessing the accuracy of prediction algorithms for classification: an overview," *Bioinformatics*, vol. 16, no. 5, pp. 412–424, 05 2000.

[20] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi, "An empirical study of just-in-time defect prediction using cross-project models," in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14)*, 2014, pp. 172–181.

[21] Q. Huang, X. Xia, and D. Lo, "Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction," *Empirical Software Engineering*, vol. 24, no. 5, pp. 2823–2862, 2019.

[22] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, and S. Li, "Just-in-time defect identification and localization: A two-phase framework," *IEEE Transactions on Software Engineering*, pp. 1–20, 2020.