Issue-Driven Features for Software Fault Prediction

Amir Elmishali and **Meir Kalech**
Software and Information Systems Engineering
Ben Gurion University of the Negev
e-mails: amir9979@gmail.com , kalech@bgu.ac.il

Abstract

Nowadays, software systems are an essential component of any modern industry. Unfortunately, the more complex software gets, the more likely it is to fail. A promising strategy is to use fault prediction models to predict which components may be faulty. Features are key factors to the success of the prediction, and thus extracting significant features can improve the model's accuracy. In the literature, software metrics are used as features to construct fault prediction models. A fault occurs when the software behaves differently than it is required to. However, software metrics are designed to measure the developed code rather than the requirements it meets. In this paper we present a novel paradigm to construct features that combine the software metrics as well as the details of the requirements, we call it *Issue-Driven features*. Evaluation, conducted on 86 open source projects, shows that issue-driven features are more accurate than state-of-the-art features.

1 Introduction

Software's significance, as well as its complexity, is practically growing in almost every field of our lives. The growing complexity of software leads to software failures that are more difficult to resolve. Unfortunately, software failures are common, and their impact can be significant and costly. Early detection of faults may lead to timely correction of these faults and delivery of maintainable software. Therefore, many studies in software engineering and artificial intelligence have been focused on approaches for finding faulty code in the early phases of software development life cycle. In particular, one of those approaches is fault prediction that implements prediction models to estimate which software components are faulty.

Software Fault Prediction (SFP) is very important and essential to improve the software quality and reduce the maintenance effort before the system is deployed. SFP enables to classify software components as fault-prone or healthy. SFP model is constructed using various software metrics and fault information of previous releases of the project system. In particular, a training set is generated by extracting a set of features from each software component and assigning a target label to the component indicating whether it is faulty or healthy. Then this training set is fed to a machine learning algorithm, which generates a classification model. This model is used to predict whether or not the components in the next version will be faulty. SFP models have been proved in the literature to produce classifiers with a great predictive performance [1] and to improve bug localization process [2].

Selecting the features that best describe each component has a great influence on the accuracy of the classification model. Previous research in SFP proposed to use software metrics as indicators for fault prediction task. These metrics have been originally created to measure various properties of the developed code such as code size and complexity, object oriented metrics and process metrics [3; 4]. However, measuring only the code properties to predict faults is not sufficient since a fault occurs when a software component behaves unexpectedly. The expected behaviour of a component is derived from the requirements of the system. Therefore, the features for SFP should represent the code properties alongside the requirements properties of the component.

A modern software developers team uses a version control system (such as GIT) to manage the modifications in the code and an issue tracking system (such as Jira and Bugzilla) to record and maintain the requested and planned tasks need to be done in the system, such as reported bugs and new features. An issue in the system is a report of a specific task, its status and other relevant information.

We can map each given requirement to the software

components that fulfilled it by monitoring issues and code changes.

The goal of this paper is to propose a novel paradigm to construct features that combine the software metrics and the issued task details, we call it *Issue-Driven features*. These features are designed to overcome the gap between the developed software metrics and the task it expected to complete. We empirically study the impact of issue-driven features on fault prediction models and we compare our results with known feature sets used in the literature.

Given the research goal, we mined 86 software repositories and extracted the issue-driven features for the SFP problem, as well as other features sets used in the literature. We evaluate the performance of SFP model generated with each set and show that issue-driven features outperform all other features sets. In addition, we show that issue-driven features are the most important features, in terms of Gini importance

2 Related Work

Software fault prediction (SFP) is one of the most actively researched areas in software engineering. SFP is the process of developing models that can be used by the developers teams to detect faulty components such as methods or classes in the early phases of software development life cycle. The prediction model performance is influenced by modeling techniques and metrics. The choice of which modeling technique to use has lesser impact on the model accuracy. On the other hand, the choice of which software metric set to use has greater impact on the model accuracy. Several literature reviews analyzed and categorized the software metrics used for SFP. [3] divided software metrics into three categories based on the goal of the metrics and the time there been proposed. First, *traditional metrics* are metrics that aimed to measure the size and complexity of a code. Second, *object oriented metrics* are metrics that aim to capture the object oriented properties such as cohesion, coupling and inheritance. Third, are *process metrics* that measure the quality of the development process such as number of changes and number of bugs. Rathore et. al. [4] categorized the metrics into two classes according to the way they are extracted. First are *product metrics* computed on a finally developed software product and includes the traditional, object oriented and dynamic metrics. Second are *process metrics* that are collected across the software development life cycle. We explain the features sets deeply in Section 4. Based on the literature reviews of Kumar et. al. and Radjenovic [5; 3], the most commonly used software metrics suites are: Chidamber and Kemerer, Abreu MOOD metrics suite, Bieman and Kang, Briand et al., Halstead, Lorenz and Kidd and McCabe [6; 7; 8; 9].

3 Problem Definition and Methodology

3.1 Problem Definition

Fault prediction is a classification problem. Given a component, the goal is to determine its class, healthy or faulty. Software fault prediction is described as the task of predicting which components in the next version of the software contain a defect. Supervised machine learning algorithms are commonly used to solve classification problems. They receive as input a training set, which is fed into a classification algorithm. This set is composed by a set of instances, in our domain these are software components, and their correct labeling, i.e., the correct class - healthy or faulty - for each instance. Then, they output a classification model, which maps a new instance to a class.

3.2 Generating a Fault Prediction Model

To build a fault prediction model we require a training set. In our case, the training set is composed of components from previous versions of the software under analysis. Version control systems, like Git and Mercurial, track modifications done to the source files and record the state of every released version. Therefore, by analyzing version control systems, we can train a fault prediction model using components from previous versions, and evaluate the prediction model with the components from the next version.

The label of an instance is whether the file is faulty or not. Issue tracking systems such as Jira and Bugzilla, record all reported bugs and track changes in their status, including when a bug gets fixed. A key feature in modern issue tracking and version control systems is that they enable to track which modifications in the source were performed in order to fix a specific bug. Formally, given a bug X we $\Phi(X)$ is a function that returns a set of source files in the version control system that were modified to fix bug X. To implement function $\Phi(X)$, we start by extracting closed bug reports that refer to the version under analysis from the issue tracking system. Then, we map each bug to the commit that fixed it. To this end, we use the id of the bug issue, and search the corresponding issue id in all of the commit messages of the version under analyzing. After matching each bug issue to the respective bug fixing commit, we label the changed files in the commit as defective if they were changed in that particular fixing commit. Consequently, for each version, the files labeled as defective are the ones that were previously changed at least once in a fixing commit.

4 Feature Extraction

One of the key requirements to achieve good performance while predicting the target state is to choose meaningful features. Many possible features were proposed in the literature for software fault prediction task. In particular, software metrics are well known features

to be used for this task. Software metrics have been introduced to estimate the quality of the software artifacts currently under development for an effective and efficient software quality assurance process, By using metrics, a software project can be quantitatively analyzed and its quality can be evaluated. Generally, each software metric is related to some functional properties of the software project such as coupling, cohesion, inheritance, code change, etc., and is used to indicate an external quality attribute such as reliability, testability, and fault-proneness.

Rathore et. al.[4] survey the software metrics used by existing software fault prediction models and divide them into product features and process features. Product features aim to measure the final developed product and can be categorized into three feature sets: traditional, Object-Oriented and dynamic features. Process metrics aim to measure the quality of the development process and software life cycle.

- *Product features* - product metrics are calculated using various features of finally developed software product. These metrics are generally used to check whether a software product confirms certain norms or code conventions. Broadly, product metrics can be classified as traditional metrics, object-oriented metrics, and dynamic metrics.
  - *Traditional features* - These features, designed during the initial days of emergence of software engineering, are known as traditional metrics. they mainly include size and complexity metrics such as number lines of code and number of functions.
  - *Object oriented features* - These features are software complexity metrics that are specifically designed for object oriented programs. They include metrics like cohesion and coupling levels and depth of inheritance.
  - *Dynamic features* - Dynamic metrics refer to the set of metrics which depend on the features gathered from a program execution. These metrics reveal the behavior of the software components during execution, and are used to measure specific run time properties of programs, components, and systems. On the contrary to the static metrics that are calculated from static non-executing models, the dynamic metrics are used to identify the objects that are the most coupled and/or complex during run-time. These metrics provide different indication on the quality of the design.
- *Process features* - Process features refer to the set of metrics that depend on the features collected across the software development life cycle. For instance, the number of times a component has been modified, the time passed from the last modification, etc. In contrast to product metrics, process features were designed to measure the quality of the development process instead of the final product. They help to provide a set of process measures that lead to long-term software process improvement.

It is not clear from the literature which combination of features yields the most accurate fault predictions. Therefore, in our experiments we use features from all the sets.

5 Issues Driven Features

Nowadays, software development teams manage their day-to-day tasks using issue tracking system like Jira, Bugzilla etc. Issue tracking systems record issues that should be implemented in the system such as bugs to be fixed and new features to develop. An issue details the status of the task, the type of the task, a literal explanation of the task and the task priority. To resolve an issue, a developer adds a commit that completes the issue task. Software bugs occur when the code does not perform the issued task correctly. Most of the software metrics measure the quality based on the code properties solely rather than the desired tasks they seek to accomplish. We suggest combining issues and code properties to calculate code metrics as a function of the issues properties that they address. We call these new features **"issue-driven features"**. An issue can be represented by its: (1) type (bug fix, improvement or new feature), (2) priority (major, minor, trivial), and (3) severity (blocker, normal, enhancement). Note that we can easily extract these fields from the issue tracking system. Then, we map the issue to the developed component by analyzing the changes in the GIT commits that the developer added to resolve this issue. Figure 1 shows an example of an issue report CAMEL-12078 (upper) and a commit that resolved it (lower). As a result, for each issue we have the changes have been done to resolve it. For example, one of the features that we propose is to measure the added complexity of a change for different issues type. As shown in Figure 1, we can analyze the changed file and mark the added "else" statement as increase of the code complexity that fixed the bug. To demonstrate the effectiveness of the issue-driven features, we show how to empower the known features, process and product, by adding the issue information.

- *Process features:* The process features are calculated as an aggregation of the change metrics on the relevant commits e.g. the number of line insertions for a file. In order to add the issue's information, we extract the issue that was resolved by each commit and record the changes have been done in the commit to solve the issue. For example, we calculate the number of line insertions that fixed a bug.
- *Product features:* A product feature is extracted by analyzing the source lines of the component. For

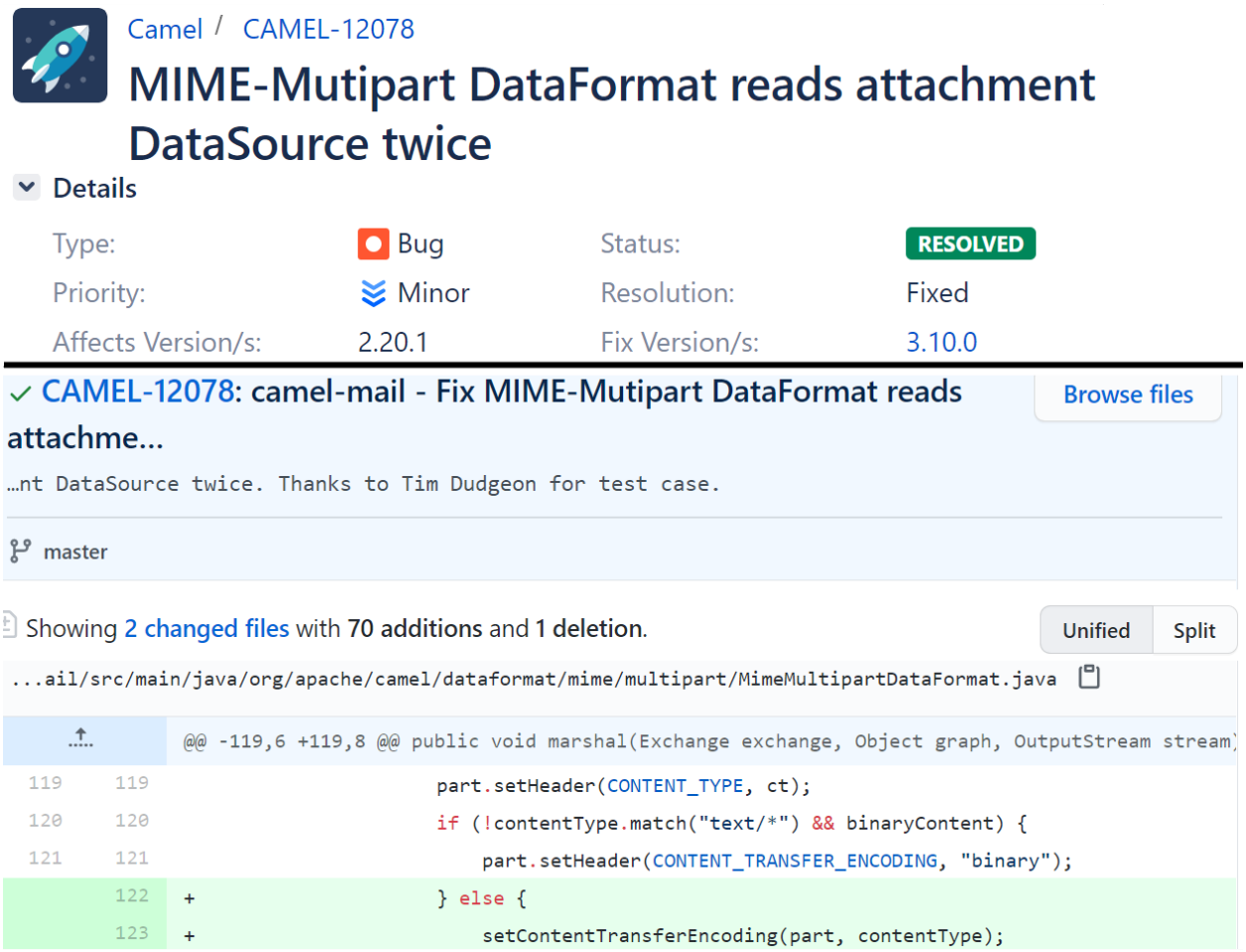

Figure 1: A screenshot of a bug report number CAMEL-12078 (upper) and the commit that resolved it (lower). We can see that a new 'else' statement added in order to resolve the bug.

example, the number of function calls in a component. In order to add the issue's information for the product features, we annotate for each source line the latest commit that modified it. Then, we use our mapping in order to find the issue that has been resolved by the commit. Finally, we calculate each product metric separately for the different values of the issue type, priority and severity. For example, to get the number of function calls in for issue type "bug" we sum up the number of calls only for source line that have been modified in commits mapped to issue of type "bug".

Next we demonstrate the issue-driven features extraction with the product metric *lines of source code (LOC)*. First we map each source line to the issue that has been resolved by the last commit that changed the line. Figure 2 shows a screenshot of function *createMixedMultipartAttachments* from Apache Camel project. For each source line we mention the issue that mapped to the line and the type of each issue (bug/improvement/new feature). The LOC of bug issues is 5, LOC of improvement issues is 4 and LOC of new features is 2.

To demonstrate a issue-driven process feature extraction, we focus on the total number of modification (MOD) of the commit. Here we derive the MOD per issue type. For simplicity, we focus only on the three commits in the square. There is a vertical line next to each commit, that marks the source lines that have been modified by the commit. We can see that the MOD value is 11 and the MOD of the bug issues, improvement issues and new feature issues, are 5, 4, and 2 respectively. In the experiment section we thoroughly explain which features we extracted.

6 Evaluation

Our research goal is to present the effectiveness of the *issue-driven features* for software fault prediction. Therefore, we designed our study to empirically com-

Figure 2: A screenshot of the function *createMixedMultipartAttachments* from Apache Camel project. For each source line we show the issue resolved by the last commit that changed the line. Also, we add the issue type of each issue, where "Bug", "Iml" and "New" represent a bug fix, improvement and new feature, respectively. The highlighted square shows the lines that were modified by each commit. First, CAMEL-385's commit added 5 lines, then, CAMEL-1645's commit modified 4 and finally, CAMEL-14578's commit modified the last two.

pare the performance of fault prediction models trained with the issue-driven features against other feature sets proposed in the literature. We report an experimental study designed to address the following research questions.

RQ1. *Do issue-driven* **product** *features perform better than other product features proposed in the literature?*

RQ2. *Do issue-driven* **process** *features perform better than other process features proposed in the literature?*

RQ3. *Which features influence the most on the accuracy of the fault prediction model?*

6.1 Experimental Setup

We start by collecting the data from repositories, which includes metrics and defects information. Then, we apply feature extraction whose purpose is to extract the features and organize them in sets. Next, we train classification models to predict defects based on several algorithms and optimise them with hyper-parameterization. Last, we cross-validate the models and evaluate them using different classification metrics. Each step is represented in the following subsections.

Data Collection The first step of our approach is to collect the data and to generate the datasets required for training and testing of the classifiers. We evaluate our approach 86 projects from the open source organizations Apache [1] and Spring [2] written in Java that managed their source code using Git version control system and an issue tracking system (JIRA or BUGZILLA). We filtered the projects as follows. First, we filtered out projects without reported resolved bugs or less than 5 released versions. Then we iterated the resolved bugs and mapped them to the commits that resolved them. Next, for each version we labeled the faulty files in the version if they changed in a commit in the version that resolved a bug. Finally, for each project, we filtered out versions with faulty files' ratio lower than 5% and higher than 30%, since it composes a good representation of bugs that reduces the class imbalance, that is produced by the low number of defects, and it is not an outlier, for instance, a version that was created just to fix issues. For each project we selected 4 version as a training set and a later version as a test set.

Feature Extraction It is not clear from the literature which combination of features return the best fault prediction model. Therefore, we extracted commonly used feature sets from the literature [4; 3], consist of both process and product features. We implemented 122 features including the following product features sets:

1. Chidamber and Kemerer metrics suite [6] (CK),
2. Halstead complexity metrics [8] (HALSTEAD),

[1] https://www.apache.org/
[2] https://spring.io/

3. Lorenz and Kidd OO features [9] (LK),
4. MOOD features [7] (MOOD).

In addition, we implemented 17 process features including the following:

1. code delta and change metrics (such as number of changes and type of changes) [10],
2. Time difference between commits [11; 12]
3. developer based features [11; 13].

Training Classifiers Several learning algorithms were considered to generate the fault prediction model: Random Forest, XGB Classifier and Gradient Boosting Classifier. Preliminary comparison found that the Random Forest learning algorithm with 1000 estimators performs best with our datasets. The depth of the trees was limited to five and the function to measure the quality of a split was set to Gini. We used 10 fold cross validation Random Forest for the rest of the experiments.

6.2 Data Analysis and Metrics

To evaluate the fault prediction models, we followed Rathore et al. [4] literature review that recommends to use precision, recall and the area under the ROC curve (AUC) as evaluation metrics. Moreover, they recommend AUC as a primary indicator. We describe the metrics in the rest of this section. Precision and recall measure the relationships between specific parameters in the confusion matrix:

$$P = \frac{TP}{TP + FP} \qquad R = \frac{TP}{TP + FN} \tag{1}$$

where,

- TP is the number of classes containing fault that were correctly predicted as faulty;
- TN is the number of healthy classes that were predicted as healthy;
- FP is the number of classes where the classifier failed to predict, by declaring healthy classes as faulty;
- FN is the number of classes where the classifier failed to predict, by declaring faulty classes as healthy;

In addition, we use Area Under the Curve (AUC) of the Receiver Operating Characteristic curve (ROC). The ROC visualizes a trade-off between the number of correctly predicted faulty modules and the number of incorrectly predicted non-faulty modules. As a result, the closest it is to 1, the better the classifier's ability to distinguish between classes that are or are not affected by the fault.

feature set	AUC	Recall	precision	importance
issue-driven process	**0.74**	**0.16**	**0.31**	0.31
process	0.68	0.14	0.30	0.07
issue-driven product	**0.75**	**0.16**	**0.33**	0.40
product	0.62	0.11	0.27	0.22

Table 1: The fault prediction performance for different feature sets. The first two rows compare between issue-driven process features and common process features used in the literature, and the next two rows compare between issue-driven product features and common product features used in the literature. The highest value of each metric is highlighted. The importance of a set is calculated as the sum of the Gini importance of the features in the set.

6.3 Results

In this section, we discuss the obtained results focusing on the research questions we initially defined. As such, we first analyse the results of the fault prediction models trained with different feature sets, and then analyse the importance of each feature set.

To address **RQ.1** and **RQ.2** we evaluated whether the performance of our models, trained with issue-driven features, outperform those trained with feature sets from the literature. Table 1 shows the arithmetic mean for all the scores representing the comparison between issue-driven features and the known features. The first two rows compare between issue-driven process features and common process features used in the literature, and the next two rows compare between issue-driven product features and common product features. The precision and especially recall results are fairly low. This is understandable, since the imbalance nature of the dataset damage the TP score of the models [14].

Regarding **RQ.1**, we can observe that the issue-driven features perform better in all metrics. This is most noticeable in the primary indicator AUC. Furthermore, for 81% (70 out of 86) of the projects the issue-driven features performed better than the other features. The significance level of the results is $p < 0.01$. Regarding **RQ.2**, we evaluate process features sets as listed at Rathure et al. [4] such as code delta, code churn and change metrics. We compare the results of a model trained with those features to a model trained with issue-driven variant of those features. Results among all metrics show that issue-driven features outperform the process features used in the literature. Furthermore, for 86% (74 out of 86) of the projects the issue-driven features performed better than other features. The significance level of the results is $p < 0.01$.

To address **RQ.3** we evaluated which feature set influence the prediction model the most, when training with all the feature sets together. To do so, we relied on the

importance score of the random forest classifier. The importance of a feature is computed as the (normalized) total reduction of the criterion brought by that feature. It is also known as the Gini importance. We computed the importance of a feature set as the sum of the importance of the features in the set. The column "importance" in Table 1 shows the arithmetic means of the importance score for each feature set. These results show that issue-driven features are the most importance features in the model.

7 Threats To Validity

For our study, we identified the following threats to validity. The projects we used for evaluation were limited to *open-source projects from Apache and Spring written in Java*. This is a threat to the generalization of our results. However, several fault prediction studies have used projects from open source projects as the software archive [15] and, in addition, projects from Apache have been integrated into the Promise data set. The use of the *issue tracking system* is also a threat to validity towards the result's generalization. However, their use is coupled with the issue tracking system of the open source projects.

8 Conclusions and Future Work

In this study we present a novel feature set for the fault prediction problem, named issue-driven features. We demonstrated how issue-driven features overcome the limitation of traditional software metrics that are agnostic to the requirements of the software. Next, we evaluated the impact of issue-driven features on 86 open source projects from two organizations. We evaluated the performance of issue-driven features against traditional features for both process and product feature classes. Moreover, we investigated the importance of the issue-driven features among all features. The results show that issue-driven features are significantly better than traditional features for both classes and achieve an improvement of 6% to 13% in terms of AUC. In future work we propose to improve Issue-Driven features using the relation between overlapping issues such as new feature and its improvements. Moreover, in future work, we plan to use the Issue-Driven feature to cross-project software fault prediction task.

References

[1] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.

[2] Amir Elmishali, Roni Stern, and Meir Kalech. Data-augmented software diagnosis. In *AAAI*, volume 16, pages 4003–4009, 2016.

[3] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and software technology*, 55(8):1397–1418, 2013.

[4] Santosh S Rathore and Sandeep Kumar. A study on software fault prediction techniques. *Artificial Intelligence Review*, 51(2):255–327, 2019.

[5] Lov Kumar, Sanjay Misra, and Santanu Ku Rath. An empirical analysis of the effectiveness of software metrics and fault prediction model for identifying faulty classes. *Computer standards & interfaces*, 53:1–32, 2017.

[6] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[7] Fernando Brito Abreu and Rogério Carapuça. Object-oriented software engineering: Measuring and controlling the development process. In *Proceedings of the 4th international conference on software quality*, volume 186, 1994.

[8] Maurice H Halstead. Elements of software science. 1977.

[9] Mark Lorenz and Jeff Kidd. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., 1994.

[10] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change bursts as defect predictors. In *2010 IEEE 21st international symposium on software reliability engineering*, pages 309–318. IEEE, 2010.

[11] Shinsuke Matsumoto, Yasutaka Kamei, Akito Monden, Ken-ichi Matsumoto, and Masahide Nakamura. An analysis of developer metrics for fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, pages 1–9, 2010.

[12] Garvit Rajesh Choudhary, Sandeep Kumar, Kuldeep Kumar, Alok Mishra, and Cagatay Catal. Empirical analysis of change metrics for software fault prediction. *Computers & Electrical Engineering*, 67:15–24, 2018.

[13] Dario Di Nucci, Fabio Palomba, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia. A developer centered bug prediction model. *IEEE Transactions on Software Engineering*, 44(1):5–24, 2017.

[14] C Arun and C Lakshmi. Class imbalance in software fault prediction data set. In *Artificial Intelligence and Evolutionary Computations in Engineering Systems*, pages 745–757. Springer, 2020.

[15] Amir Elmishali, Roni Stern, and Meir Kalech. Debguer: A tool for bug prediction and diagnosis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 9446–9451, 2019.