IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 16 January 2023, accepted 31 January 2023, date of publication 3 February 2023, date of current version 8 February 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3242045

RESEARCH ARTICLE

Empirical Study: How Issue Classification Influences Software Defect Prediction

PETAR AFRIC, DAVOR VUKADIN, MARIN SILIC, (Member, IEEE), AND GORAN DELAC, (Member, IEEE)
Faculty of Electrical Engineering and Computing, University of Zagreb, 10000 Zagreb, Croatia

Corresponding author: Marin Silic (marin.silic@fer.hr)

This work was supported by the European Regional Development Fund through the System for Detection of Malicious Transactions in Electronic Payment Operations Based on Machine Learning Research under Project IRI-II KK.01.2.1.02.0192, in part by the VODIME—The Waters of Imotski Region Research under Project KK.05.1.1.02.0024, and in part by the Croatian Science Foundation through the Reliable Composite Applications Based on Web Services Research Project under HRZZ-IP-01-2018-6423.

ABSTRACT Software defect prediction aims to identify potentially defective software modules to better allocate limited quality assurance resources. Practitioners often do this by utilizing supervised models trained using historical data. This data is gathered by mining version control and issue tracking systems. Version control commits are linked to issues they address. If the linked issue is classified as a bug report, the change is considered as bug fixing. The problem arises from the fact that issues are often incorrectly classified within issue tracking systems. This introduces noise into the gathered datasets. In this paper, we investigate the influence issue classification has on software defect prediction dataset quality and resulting model performance. To do this, we mine data from 7 popular open-source repositories, create issue classification and software defect prediction datasets for each of them. We investigate issue classification using four different methods; a simple keyword heuristic, an improved keyword heuristic, the FastText model and the RoBERTa model. Our results show that using the RoBERTa model for issue classification produces the best software defect prediction datasets, containing on average 14.3641% of mislabeled instances. SDP models trained on such datasets achieve superior performance, to those trained on SDP datasets created using other issue classification methods, in 65 out of 84 experiments, with 55 of them being statistically relevant. Furthermore, in 17 out of 28 experiments we could not show a statistically relevant performance difference between SDP models trained on RoBERTa derived software defect prediction datasets and those created using manually labeled issues.

INDEX TERMS Issue tracking, version control systems, natural language processing, issue classification, software defect prediction, RoBERTa.

I. INTRODUCTION

Software defect prediction (SDP) is a popular topic in software engineering research. The aim of SDP is to identify potentially defective software modules. This information can be used by quality assurance (QA) teams to better allocate their limited resources and thus further improve the quality of developed software products [2]. This is an important activity since researchers estimate that software defects and poor QA have cost the US industry $2.08 trillion in 2020 alone [1].

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

SDP is a highly active research field. However, some of the obtained results could be misleading due to datasets used to conduct studies. Researchers have pointed out that datasets created by mining software repositories might contain a substantial amount of noise. When creating an SDP dataset, researchers inspect commit messages in an effort to identify links to issues defined in the issue tracking system. Classes assigned to issues are then used to derive the labels of the created SDP datasets. Issue classes specify the type of issue, for example, a modification request, a feature request, or a bug report.

In this process, there are two situations where noise can arise. First, it can arise when commits are not

successfully linked to issues. Second, it can arise when issues in the issue tracking system are incorrectly classified. The introduced noise can result in biased and unreliable results. Researchers rely on issue classification. For example, Zimmermann et al. [33] constructed the *Eclipse Bug Data Repository* identifying defective code by searching for bug report references and keywords such as "*fixed*" and "*bug*". A similar approach was used by Cubranic and Murphy [88], Fischer et al. [89], Sliwerski et al. [8] and Bachmann et al. [90].

Herzig et al. [9] examined a substantial number of issues and found 33.8% of all bug reports were incorrectly classified. They concluded that users are the ones performing issue classification and are the source of incorrect classifications. If the software does not meet user expectations, they tend to raise an issue, and classify it as a bug report. However, users often lack technical knowledge, and insight into project details, which results in incorrect classifications. Developers could correct the issue classification, but there is no incentive for them to do so.

Kim et al. [10], Seiffert et al. [15], Pandey and Tripathi [16], Tantithamthavorn et al. [17] have all pointed out that noise in the resulting dataset can lead to severe degradation of model performance. While Khan et al. [18] showed that noise filters struggle to mitigate the problem, once noise is present in the dataset.

Antoniol et al. [19] also found that issues marked as bugs might not actually refer to bugs. They proposed using issue descriptions to classify issues and thus reduce the amount of noise in derived defect prediction datasets. In their investigation they used traditional models such as Decision Trees, Naive Bayes, and Logistic Regression on issues coming from three Java based repositories. They showed that such models can achieve an F1 score of 0.70. Please note that this metric is not directly given in their paper, instead we calculated it from the evaluation data they provided in the paper.

Since their paper was published, advanced Natural Language Processing (NLP) models, such as BERT [83], have been developed. Researchers [11], [12], [13], [14] have investigated using NLP models for issue classification and obtained promising results. However, none of this research is focused on issue classification in the context of SDP. To the best of our knowledge, no research has been done investigating the impact of issue classification on SDP dataset quality and resulting model performance. In this paper, we investigate the benefits of using such models during SDP dataset creation. The effect issue classification quality has on the noise present in the resulting SDP dataset, as well as influence on the final SDP model performance.

More specifically, we created new datasets by mining popular open-source repositories. We mined data from 7 repositories, collecting all commit data and all issue tracking data. For each dataset in each commit, we identify issue references. Issues referenced by at least one commit are considered *issues-of-interest* (IOI). Commits containing at least one issue reference are considered *commits-of-interest* (COI). For every file mentioned in version control, we identify all commits it has been modified by. If all commits modifying a file are COI, meaning all of them reference an IOI, then the file is considered a *file-of-interest* (FOI). Each version of a file, meaning its state after a commit it has been modified by, is decoded, stripped of all comments, and encoded using GraphCodeBERT. These encodings are used to construct semantic features, which in addition to other process and code complexity features are used to construct a software defect prediction instance. The instance is considered defect prone if at least one commit in its history references a bug related issue. For each repository, at least 1000 IOI are manually labeled and used to create a golden SDP dataset. We then investigate the amount of noise induced into the SDP dataset if the issues are not manually labeled, but instead automatically determined using a keyword matching heuristic (KWM), an improved keyword matching heuristic (IKWM), a FastText model and finally a RoBERTa model. For each resulting SDP dataset we train Logistic Regression, Decision Trees, Naive Bayes and K-Nearest Neighbours models and investigate the impact of noise in the SDP dataset on their performance.

In simpler terms, we constructed file level SDP datasets. The labels of these datasets are determined based on issue classification. If a source code file is edited by a commit, linking to a bug related issue then that file is considered defect prone, otherwise it is considered not defect prone. Depending on the quality of the issue classification, we investigate the amount of noise induced in the resulting SDP datasets and the effect this has on SDP model performance. We tested out four different issue classification methods. A KWM model, an IKWM model, a FastText model and a RoBERTa model. As SDP models, we trained Logistic Regression, Decision Trees, Naive Bayes and K-Nearest Neighbours models. These models predict if a source code file is defect prone or not, meaning we investigate how their performance is impacted by the amount of noise in the SDP dataset, and the amount of noise is a direct consequence of the issue classification quality.

Our results show that using the RoBERTa model for issue classification produced the fewest mislabeled instances in SDP datasets compared to other approaches. These datasets contain an average of 14.3641% mislabeled instances. We compared, and statistically validated, the performance of models trained on such SDP datasets and on those created using other methods. In 65 out of 84 experiments models trained on SDP datasets created using RoBERTa had superior performance and they had statistically relevant superior performance in 55 out of 84 experiments which is equal to 65.4761% of the time. Further, we compared the performance of models trained on RoBERTa derived SDP datasets to those trained on SDP datasets derived from manual issue classification. In 17 out of 28 experiments

we could not show a statistically relevant performance difference.

In summary, this work makes the following contributions:

- Fully mined commit and issue data for 7 popular open-source repositories. Manually labeled issue classification datasets consisting of at least 1000 issues for each repository and derived software defect prediction datasets.
- Experiments which investigates the performance of simple keywords matching heuristics (KWM), improved keyword matching heuristics (IKWM), FastText model, and RoBERTa model on issue classification and the resulting impact on SDP datasets noise levels and model performance.

The rest of this paper is organized as follows. Sect. II gives a more detailed introduction to SDP and an overview of related work. Sect. III describes how data is collected and how data of interest is identified. Sect. IV describes how issue classification datasets are created and how issue classification models are developed. Sect. V describes how SDP datasets are created and which models are used for final SDP classification. Sect. VI presents the obtained results. Sect. VII outlines threats to the validity of this study. At the end, Sect. VIII concludes this research with the author's final remarks.

II. BACKGROUND AND RELATED WORK

In this section we provide a general overview of software defect prediction (SDP) and an introduction to related work important for the topic of this research paper. The first subsection presents the basics of SDP. The second subsection presents often used SDP metrics, talks about prediction granularity, common datasets, and common approaches to tackling the SDP problem. The third subsection introduces related work focusing on noise in SDP datasets. The fourth subsection presents Natural language processing (NLP) related work and previous work done around issue classification. Finally, the last subsection ties in this manuscript within the existing body of work.

A. SOFTWARE DEFECT PREDICTION

The field of software defect prediction (SDP) was started in 1971 by Akiyama when he published a paper investigating the relation between code complexity and the number of software defects [28]. He used *Lines of code* (LOC) as a measure of code complexity and showed that there exists a positive correlation between the two.

Through the years the field has developed and branched out. Today, SDP can be divided into within-project prediction and cross-project prediction. Utilizing project data to predict defects in that same project is called within-project prediction [91]. This can be further divided into within-version-within-project prediction and cross-version-within-project prediction, depending on whether the train and test data come from the same version or different versions of that project [91]. The first studies in the field were based on within-project prediction. However, researchers pointed to potential benefits of training a model on data derived from one project and using it to detect defects in another project [50]. This would allow completely new projects to use defect-prediction models and thus improve their QA. This approach is called cross-project prediction [72]. Zimmermann et al. [50] showed that cross-project prediction is a challenging task. They showed that it is not easy to identify on which project a model should be trained in order to perform well on another project. The basic premise of machine learning states that the training and test data are sampled from the same distribution. The fact that models trained on one project data do not perform well on another project implies that data distributions between projects differ to an extent that the basic premise of machine learning is no longer satisfied. Notable approaches for improving cross-project prediction performance are: Metric compensation [58], [59], Nearest neighbor filtering [60], Meta learning [57] and Transfer component analysis [53], [54], [55], [56]. Recently, researchers have proposed methods which would allow using projects described with different metrics to achieve cross-project defect prediction, thus increasing the amount of available train data. This is called heterogeneous defect prediction [51], [52].

B. METRICS, PREDICTION GRANULARITY AND APPROACHES TO SDP

Initially, most SDP studies were focused on defining useful metrics. McCabe [38] proposed a set of code complexity metrics based on cyclomatic complexity, McCabe complexity, and structural complexity. Halstead [39] proposed a set of complexity metrics based on operator and operand counts. Hitz and Montazeri [40] proposed code complexity metrics for object-oriented programming (OOP). Over time many different metrics have been proposed. Process metrics [41], [42], [43], [44], [45], change metrics [46], [47], [48], [49], semantic metrics [24], [25], [26], [27] and others.

GraphCodeBERT [27] is of special interest to us. GraphCodeBERT is a BERT base model for generating code representations based on code structure and data flow. It is a multi-lingual model supporting the following programming languages: Python, Java, JavaScript, PHP, Ruby and Go. In this paper we use this model to generate semantic features for created SDP instances.

Having many metrics has motivated studies concerning feature selection [61], [62] and normalization [55].

Some of these metrics can be applied at various levels of granularity and some are specific to a certain level. Researchers have performed SDP on many distinct levels: component [71], file [68], class [70], method [68] and change level [69]. Change level prediction is also called Just-In-Time (JIT) defect prediction. More granular prediction facilitates faster defect localization while less granular approaches are better suited for QA resource allocation [2].

To make studies comparable researchers have made their datasets available in public repositories. Famous examples of such repositories include *NASA* [35], [36], [37], *PROMISE* [34], *Eclipse Bug Data Repository* [33], *ReLink* [29], *A Unified Bug Dataset for Java* [31], [32] and many others.

Many kinds of models have been used to tackle SDP. Researchers have proposed using supervised models [63], [64], [65], semi-supervised models [66], [67], unsupervised models [23], tasks specific models such as BugCache [22] and even approaching the problem as an anomaly detection problem [20], [21].

C. NOISE IN SDP DATASETS

SDP datasets are created by mining version control and issue tracking systems. Each commit in the version control system is examined and an attempt is made to link it to an issue in the issue tracking system. If a link is found and the issue is marked as a bug report the state of the code prior to the changes can be considered as defective while the code after the changes can be considered as non-defective. Alternatively, this code in all its states can be considered defect prone. Another approach is to search for an earlier commit which caused the defective code and label it as a defect inducing commit. This approach is used for creating JIT defect prediction datasets. A link is considered found if a number matching an issue number is contained in the commit message, and certainty about the link is increased if it is near important keywords such as "*bug*" or "*fix*" [8], [33], [88], [89].

This process has two critical points at which noise can be introduced into the resulting dataset. First, it can be introduced if the links between a commit and an issue cannot be established. The second point where noise can be induced into the created dataset is the incorrect classification of issues in the issue tracking system [17].

Bird et al. [5] found that the number of fixed bugs does not match the number of bug issues leading to a high false negative rate. This is a consequence of mining being based on keyword matching [6], [7], [8], [33], [88], [89], [90], while developers often do not write specific keywords.

Wu et al. [29] proposed an approach called *ReLink* in order to alleviate the problem of missing issue links. They manually inspected links with explicit bug IDs in change logs and observed that the links exhibit certain commonalities. Based on these, they proposed an automatic link recovery algorithm which would automatically learn criteria of features from explicit links to recover missing links.

This motivated other studies such as *MLink* from Nguyen et al. [30] where the authors propose a multi-layered approach that considers both textual and source code features of modified code. The approach is capable of learning relations between terms in bug reports and the entity names in source code which allows it to established bug-to-fix links even when there is not much textual similarity between the two.

Herzig et al. [9] manually examined more than 7,000 issue reports from the bug databases of five open-source repositories and found 33.8% of all bug reports were incorrectly classified and an average of 39% of files marked as defective never had a bug.

Kim et al. [10] found that prediction performance decreases significantly when the dataset holds 20% - 35% of both false positive (FP) and false negative (FN) and proposed a noise detection and elimination algorithm. Seiffert et al. [15] did an extensive study of class imbalance and dataset noise effects. Their results correspond to those presented by Kim et al. [10].

Pandey and Tripathi [16] performed an empirical study focused on dealing with noise and class imbalance issues in software defect prediction. They show that if a dataset contains 10% - 40% of incorrectly labeled instances the true positive rate (TPR) and true negative rate (TNR) are reduced by 20% - 30% and receiver operating characteristic (ROC) values are reduced by 40% - 50%.

Tantithamthavorn et al. [17] investigate the effects of noise caused by incorrect issue classification on the performance of SDP models. In their study they used 3931 manually labeled issues derived from Apache Jackrabbit and Lucene systems. Based on the obtained results they point out that incorrect issue classification does not occur completely randomly. They show that non-random noise does not degrade the model precision but can degrade recall by 32% - 44%.

Khan et al. [18] investigate the effects of 9 different noise filters for dealing with incorrect instance labels. Instead of randomly generated noise, they use a dataset with clean labels annotated by experts and noisy labels obtained by heuristics. They observe that noise filters mostly struggle to improve the performance over noisy data.

Antoniol et al. [19] found that bug reports might refer to perfective and adaptive maintenance, refactoring, discussions, or requests for help. They carried out their experiment on three repositories; Mozilla, Eclipse, and JBoss and showed that decision trees, naive Bayes classifiers, and logistic regression models can classify issues based on their descriptions achieving an F1 score of 0.70.

D. NATURAL LANGUAGE PROCESSING AND ISSUE CLASSIFICATION

Earlier work in the area of Natuaral language processing (NLP) focused on word2vec models [74], convolutional models [75], [76], [77] recurrent models [78], [79] and, more recently, attention-based models [80], [81]. Substantial earlier work has shown that pre-trained models on large corpora are beneficial for text classification and other NLP tasks [82]. Using pre-trained models offers the benefit of avoiding model training from scratch, thus, speeding up the fine-tuning process and producing higher performance models than those trained only on one specific task.

FastText [73] is a word embedding method which is an extension of the word2vec model. It is considered a bag of words model. Instead of learning each word directly it represents each word as a set on n-grams. In its original publication it was shown to be much faster than the deep models of that time, with comparative performance.

The current widely adopted language models are BERT [83] and RoBERTa [84]. Both models offer the same architectural design, using the encoder part of the multi-layer bidirectional Transformer architecture [85] which was pre-trained on large text corpora - the BooksCorpus (800M words) [86] and English Wikipedia (2,500M words). The Cloze task [87] inspired the masked language model (MLM) objective used to train BERT and RoBERTa in conjunction with the next sentence prediction task. The method of using a large pre-trained model and fine-tuning in downstream tasks has made a breakthrough in several natural language understanding tasks [83], [84].

Researchers [11], [12], [13], [14] have looked into issue classification with various motivations.

Wang et al. [11] used BERT to recommend GitHub labels based on issue descriptions. Since issue creators often do not label issues, it is left to repository maintainers to label them which can become very time consuming, thus accurate automatic labeling would help reduce the amount of necessary manual labor.

Herbold et al. [12] stress that reported issue types often do not match the description of the issue. They attempt to improve upon existing issue classification by incorporating manually specified knowledge about issues.

Siddiq and Santos [13] propose TICKET TAGGER which can be used to automatically assign labels to GitHub issues. Again, the main goal is to reduce the amount of necessary manual issue labeling.

Siddiq et al. [14] apply BERT to GitHub issue classification and compare its performance to FastText. They point out that developers can find it difficult to manually label issues and thus would benefit from automatic issue labeling.

To the best of our knowledge the approach of using RoBERTa for issue classification in SDP dataset creation remains unexplored. Although, it has shown promising results in issue classification it is unknown if its performance reduces the noise levels of resulting SDP datasets to acceptable levels.

III. DATA COLLECTION

To acquire representative data of open-source repositories we have decided to extract information from GitHub. GitHub provides Internet hosting for software development and version control. At the time of writing (November 2022) it hosts 83 mil. developers and 28 mil. public repositories.

For mining repository data, we wrote a script using the *PyGithub*[1] Python package. The package is a wrapper, enabling simple usage of the GitHub API.

[1]https://pypi.org/project/PyGithub/

We queried 10 most popular repositories which have at least 5 *good first issues*. Maintainers of open-source repositories mark certain issues as *good first issues*, thus indicating that they are simple enough to be addressed by someone without previous knowledge about the project. Such issues are considered a good first step to start contributing to a repository. The presence of such issues implies an active project with a lively community, so we included it as a constraint. We filtered out repositories which are written in a programming language not supported by GraphCodeBERT, have less than 50 *stars* or less than 100 issues. Again, these constraints are included hoping to obtain only active and relevant repositories.

Information gathered for each repository is shown in Table 1. From each repository, which satisfies the previously listed criteria, we pulled issue data shown in Table 2 and commit data shown in 3. For each file changed by a commit we pulled file data shown in Table 4. All mined data is stored in JSON format.

TABLE 1. Mined repository data.

Data attribute	Description
Name	Name of the repository
Star	Number of stars awarded to the repository
Language	Programming language used in the repository
Issues	List of issues associated with the repository
Commits	List of commits associated with the repository

TABLE 2. Mined issue data.

Data attribute	Description
Id	Unique identifier
Number	In repository identifier
Title	Issue title providing a short issue description
Body	Full issue description
URL	URL to issue
Labels	Repository labels associated with the issue
Comment count	Number of comments on the issue
Has Pull Request	Was a pull request raised based on this issue

TABLE 3. Mined commit data.

Data attribute	Description
SHA	Unique identifier
Message	Commit message
Date	Time it was submitted
Files	List of files modified by the commit

Using the described procedure, we ended up with 7 repositories totaling 299773 issues and 153664 commits. Table 5 presents an alphabetically sorted list of all mined repositories,

TABLE 4. Mined file data.

Data attribute	Description
SHA	Unique identifier of the file and its specific version
Name	Fully qualified name
Change Count	Number of lines changed in the commit
Add Count	Number of lines added in the commit
Delete Count	Number of lines removed by the commit
Content	Content of the file after the commit (Encoded)
Content encoding	Encoding used for file content

TABLE 5. Alphabetically sorted mined repositories.

Name	Short name	Language	Issue count	Commit count
facebook/ react-native	react-native	JavaScript	34402	25473
huggingface/ transformers	transformers	Python	19163	10789
kubernetes/ kubernetes	kubernetes	Go	112661	28557
mui/ material-ui	material-ui	JavaScript	33815	20518
nodejs/ node	node	JavaScript	43374	37195
vercel/ next.js	next.js	JavaScript	26399	12521
ytdl-org/ youtube-dl	youtube-dl	Python	29959	18611

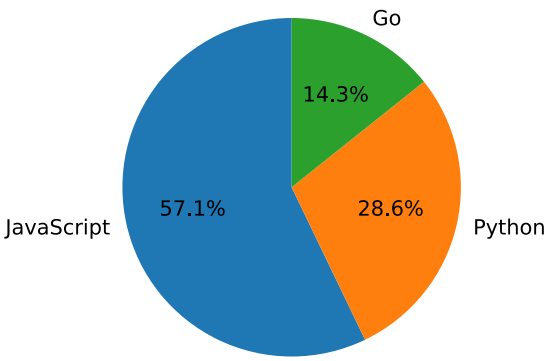


FIGURE 1. Languages used by mined repositories.

their main programming language, and the number of issues and commits from each repository. Figure 1 shows the distribution of programming languages of mined repositories.

We then analyzed each commit in each repository by applying the regular expression shown in Equation 1 to the commit message. In this way, we identify issue references. The regular expression searches for either a direct issue reference e.g., *#12345* or a link to the issue e.g., */issue/12345* or */pull/12345*. The reason we search both *issue* and *pull* is that we observed that GitHub uses both links for issues, with the only difference being that *issue* were started by someone raising an issue and *pull* were started by someone raising a pull request.

$$(? : \#|/issues/|/pull/)(d+) \tag{1}$$

Issues referenced by at least one commit are considered *issues-of-interest* (IOI). Commits having at least one issue reference in their commit message are considered *commits-of-interest* (COI). Some issues referenced in commit messages were not mined because they no longer existed on GitHub. We removed such issues from the IOI and commits mentioning them from the COI. We then inspected all commits and made a list of all files, differing them by name. For every file, we identify all commits modifying that file. If all commits modifying a file are COI, meaning all of them reference an issue, the file is considered a *file-of-interest* (FOI). We reduce the identified set of FOI by keeping only those with the following file extensions *.py*, *.php*, *.js*, *.java*, *.go* and *.rb* as those are languages supported by GraphCodeBERT. Note that this is a wider set of programming languages than those associated with the mined repositories. The repository language is the main language used in the repository, but there may still be files written in other languages. This is why we consider a wider range of languages than just the main repository language. GitHub provides the source of each version (state after commit modifying the file) of each FOI in a base64 encoded format. We decoded the remaining versions of the files and removed all comments using the *pyparsing* library. Sometimes, consecutive versions of a single file are identical, which means that the changes made between different versions were not made to the source code. Since we only want to consider changes made to the source code, we remove duplicates by keeping only the earlier versions of the file. After performing these steps, the set of IOI is reduced to only those issues that are referenced by COI of the final FOI.

Table 6 shows the final number of IOI, COI and FOI per each repository. From the data presented in Table 5 and Table 6 we can observe how some projects, such as *nodejs*, systematically reference issues in commit messages allowing many IOI, COI and FOI to be identified, while others, such as *kubernetes*, can hardly be connected.

TABLE 6. Alphabetically sorted data of interest per each repository.

Repository	Issues OI	Commits OI	Files OI
react-native	363	4443	549
transformers	3039	5567	2338
kubernetes	94	10381	195
material-ui	5408	7716	10872
node	13875	15397	15189
next.js	5501	5676	8464
youtube-dl	148	6075	72

IV. ISSUE CLASSIFICATION

In this section we describe how datasets for issue classification are created, and how issue classification models are trained based on the collected data of interest.

Figure 2 shows the simplified general overview for issue classification. In the upper part of the figure, we see issues, each consisting of a title and a description. Using these titles and descriptions, in addition to manually determined issue labels, we can train an issue classification model. In this study we consider several issue classification models. A KWM model, an IKWM model, a FastText model or a RoBERTa model. Using a trained issue classifier, we can classify new issues. This is depicted with the arrows coming out of the classifier and pointing to specific issues. The classifier assigns either a non-defective class, depicted using a small green rectangle, or a defective class, depicted using a small red rectangle. Of course, in practice different issues are used to train the model than those to which the model is applied. In the lower part of the image, we see the classified issues linking to commits which modify a yellow, blue, and red file. This part is not relevant for issue classification itself but helps understand how issues are connected to commits and consequently to source code files from which the final SDP datasets are constructed. It is important to understand this connection as the labels of SDP instances are derived from issue classification.

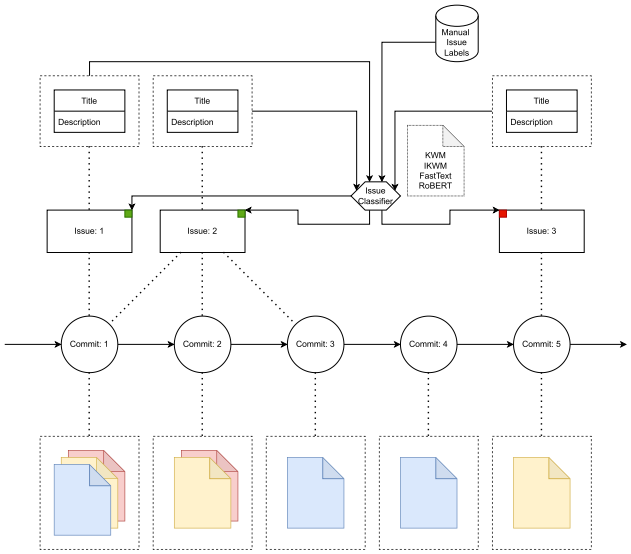


FIGURE 2. Issue classification overview.

The first subsection describes issue classification dataset creation, and the second subsection describes issue classification model development.

A. DATASET

After identifying the IOI, we manually labeled at least 1000 IOI per repository. If a repository has less than 1000 IOI, we labeled all of them. If it has more than 1000 IOI, we selected the first 1000 IOI sorted by issue number, and then added any additional IOI that were needed for FOI influenced by the first 1000 IOI.

To manually inspect and label IOI, we developed a custom web application. The application accepts JSON data

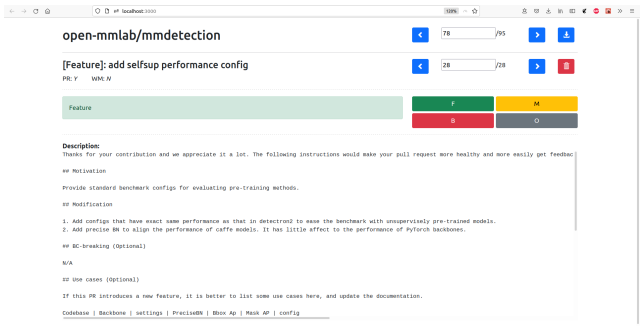


FIGURE 3. Application main view.

generated by the mining script and displays the repository name, issue title, issue description, labels associated with the issue, and whether the issue has a pull request. The application allows users to navigate through repositories and issues, label issues, delete issues, and download data in the same JSON format as the uploaded data. Figure 3 shows the application main view.

We allow issues to be labeled with one of four classes: *Feature*, *Modification*, *Bug* and *Other*. *Feature* denotes requests for new functionality. *Modification* denotes requests for change of existing functionality. *Bug* denotes reports of defective behavior of the software system. *Other* denotes questions, discussions, and other requests we could not place in any of the previous categories. However, in this study we are only interested in differing between bug related issues and non-bug related issues, thus we treat *Feature*, *Modification* and *Other* as a single label meaning *Non-Bug*. The labeling tool was made more general as we imagine other researchers might want to further distinguish issues.

To improve the labeling quality, three people labeled the IOI and the majority vote classification was taken for each issue. The content of resulting datasets per each repository are shown in Table 7. Labeled issues of interest are referred to as LIOI.

TABLE 7. Issue dataset per each repository.

Repository	Issue Count	Bug Count	Non-Bug Count	Unanimous Classification Count
react-native	363	93	270	351
transformers	1013	278	735	992
kubernetes	94	36	58	94
material-ui	1102	296	806	1086
node	1012	301	711	997
next.js	1062	328	734	1013
youtube-dl	148	55	93	148

It might be tempting to use the labels on GitHub issues as a way to classify the issues, but this approach has some problems. One issue is that the labels are not standardized or required, so they can vary significantly from one project to another, and some projects might not have any labels at all. Additionally, the labels are often used to indicate which part of the project is affected by the issue, rather than the type of issue itself.

B. MODELS

This subsection describes the development of models used for issue classification. We first describe a simple keyword matching heuristic (KWM), then an improved keyword matching heuristic, then an application of the FastText model and finally an application of the RoBERTa model.

1) SIMPLE KEYWORD MATCHING

The base model for issue classification is a simple keyword matching heuristic. It is based on a case insensitive search for *bug* or *fix* keywords. The search for these keywords was applied to the issue title and description.

2) IMPROVED KEYWORD MATCHING

The first step towards improving the KWM heuristics is to determine which keywords imply that the issue is defect related. All text is transformed into lower case, all punctuation symbols are removed and a snowball stemmer [92] is applied to each word. Issue title and description are inspected word by word, considering only words exclusively consisting of alphabetic characters and longer than 2 letters. For each word, the number of times it appears is counted (tc). Furthermore, the number of descriptions it appears in is counted (dc). For every word which occurred in a bug report, we calculate the *bug importance score* as $log(tc/dc)*(dc/bCnt)$. Similarly, for every word which occurred in a non-bug report, we calculate the *other importance score* as $log(tc/dc) * (dc/oCnt)$. Finally, for each word we subtract the *other importance score* from the *bug importance score* and sort all the words by the resulting score. By doing this we get defect implying words at one end of the list and non-bug implying words at the other end of the list. Figure 4 shows a word cloud visualization of top bug implying words and Figure 5 shows a word cloud visualization of top non-bug implying words. The described algorithm is shown in Algorithm 1.

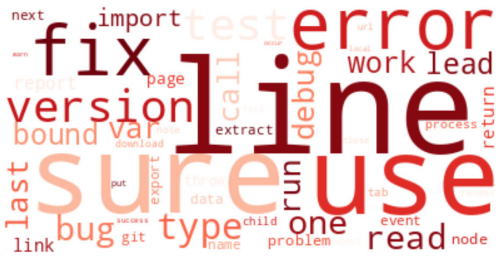


FIGURE 4. Visualization of words indicating it is a bug.

Expert knowledge was used to select a subset of keywords from the obtained list which is then used for defect related issue discovery. We inspected the list top to bottom and chose 8 meaningful words. This list of words includes: "*bug*", "*error*", "*fix*", "*issue*", "*line*", "*out*", "*not*" and "*test*.

We narrowed down the list of words we used to a small number because using too many words would have resulted in an excessively large number of possible combinations to check. For each repository, we identified the best combination

Algorithm 1 Procedure of Discovering Important Words

Input data
Issue data: *issues*
Output data
Sorted list of important words: *scores*
Procedure
$bCnt = 0$
$oCnt = 0$
$bugWC = map()$
$otherWC = map()$
$words = set()$

for *issue* in *issues* **do**
  $description = cleanText(issue["description"])$
  $description = description.lower()$
  $description = removePunctuations(description)$
  $description = applySBStemmer(description)$
  for unique *word* in *description* **do**
    if $len(word) >= 3$ and $word.isalpha()$ **then**
      $words.add(word)$
      $cnt = description.count(word)$
      if $issue["type"] == "Bug"$ **then**
        $bugWC[word].tc+ = cnt$
        $bugWC[word].dc++$
      else if $issue["type"] == "Feature"$ **then**
        $otherWC[word].tc+ = cnt$
        $otherWC[word].dc++$
      end if
    end if
  end for
  if $issue["type"] == "Bug"$ **then**
    $bCnt = bCnt + 1$
  else
    $oCnt = oCnt + 1$
  end if
end for

for *word* in *bugWC* **do**
  $tc = bugWC[word].tc$
  $dc = bugWC[word].dc$
  $bugWC[word].sc = log(tc/dc) * (dc/bCnt)$
end for
for *word* in *otherWC* **do**
  $tc = otherWC[word].tc$
  $dc = otherWC[word].dc$
  $otherWC[word].sc = log(tc/dc) * (dc/oCnt)$
end for

$scores = list()$
for *word* in *words* **do**
  $sc = bugWC[word].sc - otherWC[word].sc$
  $scores.add([sc, word])$
end for

return $sort(scores)$

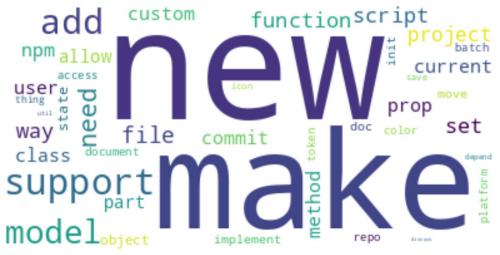


FIGURE 5. Visualization of words indicating it is not a bug.

of words to use when matching the issue title and the best combination to use when matching the issue description by testing $(2^8)^2 = 65536$ different combinations of keywords. We then used these identified keyword combinations and the previously described text pre-processing to improve our keyword matching method.

3) FastText

We use the FastText model implementation provided by the python *fasttext* library. All text is transformed into lower case, it is cleaned up by removing HTML tags, removing multiple consecutive spaces, newline spaces, tab spaces, replacing all hyperlinks with the keyword *[link]* and finally removing all non-alphanumeric characters. For each repository we train a separate FastText model treating the LIOI of that repository as the test set, and all LIOI from all other repositories as the train set. One could say that we are performing cross-project issue classification. On the train set we perform a 80/20 split taking 80% of the set as the final train set and 20% as a validation set. The model is then given 15 minutes to perform hyper-parameter optimization, on an Intel(R) Core(TM) i7-7700 @ 3.60GHz CPU, and then trained with the best identified parameters.

4) RoBERTa

Furthermore, for issue classification we use the RoBERTa model [84]. The name RoBERTa stems from *Robustly Optimized BERT Pretraining Approach* and BERT stands for *Bidirectional Encoder Representations from Transformers*. The architecture of this model is a stack of 12 transformer encoders. Each encoder consists of a multi-head attention layer and a feed forward layer with summation and normalization in between and at the encoder output. On top of this a task specific feed forward neural network and a SoftMax layer are added. Where BERT performs masking once during data pre-processing, resulting in a single static mask, RoBERTa duplicates the training data 10 times so that each sequence is masked in 10 different ways. By doing this RoBERTa avoids using the same mask for each training instance in every epoch. The resulting model is then fine-tuned for the desired task.

To prepare the issue title and description as input for the model, they are joined, all hyperlinks are replaced with the keyword *[link]*, all HTML tags are removed, as are multiple consecutive spaces, newline spaces and tab spaces. The pre-processed texts are subsequently tokenized using the model's respective tokenizer. For each repository we train a separate RoBERTa model treating the LIOI of that repository as the test set and use LIOI from all other repositories to create a train and validation set. On LIOI from other repositories, we perform an 80/20 split taking 80% of the set as the final train set and 20% as a validation set. The model receives an input length of 512 tokens, where the text that consists of fewer tokens are padded to the desired length and longer sequences are truncated to 512 tokens. Mini-batches of 4 examples are used through a 6 epoch training period. For optimization, the AdamW optimizer is applied to all model parameters with a learning rate set to $2 \cdot 10^{-5}$. The weight decay factor used in the AdamW optimizer is set to 0.001 for all model parameters except for biases, where it is set to 0. Additionally, 4 gradient accumulation steps and gradient checkpointing are used to reduce the memory footprint required for model training. Also, half precision training (*FP16*) is used to facilitate faster training and further reduce the required GPU memory. After each epoch, the model is evaluated on the validation set and at the end of training the best performing version of the model, based on results on the validation set, is selected as the definitive version of the model.

V. SOFTWARE DEFECT PREDICTION DATASETS

In this section we describe how software defect prediction dataset are derived based on the collected data of interest and determined issue classification.

Issues labeled during manual inspection are considered labeled issues of interest (LIOI). For all LIOI we identify files of interest (FOI) influenced by LIOI through the commit links using commits of interest (COI). Files influenced by LIOI are referred to as labeled files of interest (LFOI). We then proceed to create an SDP dataset for each repository.

Each LFOI results in one instance in the SDP dataset, and the label for that instance is based on the issues associated with the commits that modify the LFOI. If any of the issues is labeled as a bug related issue, we consider the instance to be bug prone, if none of the issues are bug related the instance is considered to be clean. The issue label can be determined using the manually assigned labels or those determined by the issue classification model. The features of the instance consist of three types; code complexity features, process features and semantic features. The code complexity features, or better said feature is the simplest one. For each version of a LFOI we inspect the *number of lines of code* and the feature is calculated as the *average number of lines of code* for this file through its history. The process features used are the *number of commits which modify this file*, the *absolute number of issues referenced by these commits* and the *average number of issues referenced per commit modifying this file*. Finally, the semantic features are the most complex. Each version of the LFOI is embedded using GraphCodeBERT. Given that the model input is limited to 512 tokens at a time, we encode

the source code in chunks of 512 tokens with a 256 token overlap between consecutive chunks. For each embedded chunk we take the embedding of the CLS token which should encapsulate the overall semantics of the processed chunk. We then calculate the mean CLS token over all chunks of a single version. The final semantic features for the file are the *mean CLS token of each of the file versions* and the *sum of differences of consecutive version CLS tokens*. The mean over all versions of the file should encapsulate the general propose of the source code while the sum of differences should encapsulate the file's change over time. The described code complexity, process and semantic features are concatenated to create the final features of the SDP instance. Figure 6 depicts the described process and Table 8 shows the resulting SDP datasets per each repository.

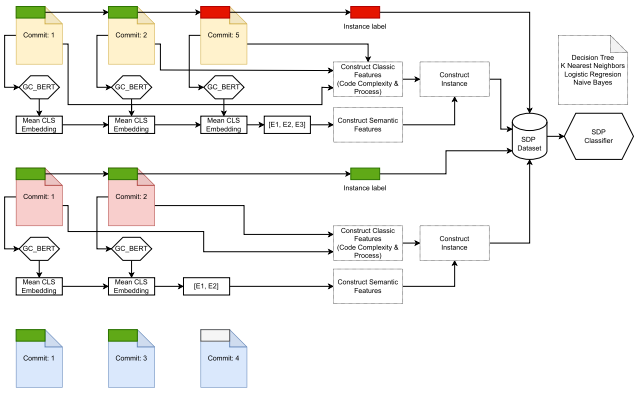


FIGURE 6. SDP overview.

TABLE 8. SDP dataset per each repository.

Repository	SDP Count	Bug Count	Non-Bug Count
react-native	549	114	435
transformers	851	312	539
kubernetes	195	91	104
material-ui	2097	1190	907
node	2188	454	1734
next.js	1484	517	967
youtube-dl	72	24	48

We trained Logistic Regression (LR), Decision Trees (DTC), Naive Bayes (NB), and K-Nearest Neighbours (KNN) models on the SDP datasets we created. We did not pursue more advanced SDP model development because the goal of the study was to investigate the impact of the quality of issue classification on the quality of the SDP dataset and the subsequent performance of the model, not to develop the most advanced SDP model. We chose to use these models because they are commonly used in SDP studies.

VI. EVALUATION

This section describes the results we obtained. For each approach and for each repository,we first present the Precision, Recall and F1 score that was achieved in the task of issue classification. Then we show the impact of issue classification on the resulting SDP dataset by presenting the confusion matrix of the automatically assigned labels compared to the "golden" labels of the SDP dataset. The confusion matrix includes four values. True Positive (TP) is the number of defective instances that were correctly identified as defective. True Negative (TN) is the number of non-defective instances that were correctly identified as non-defective. False Positive (FP) is the number of non-defective instances that were incorrectly identified as defective. False Negative (FN) is the number of defective instances that were incorrectly identified as non-defective. Finally, we train Logistic Regression, Decision Tree, Naive Bayes, and K-Nearest Neighbor models and investigate the impact of noise in the SDP dataset on their performance. Training of these models is repeated 30 times to mitigate the stochastic nature of the procedure and its influence on the obtained results. For each repository and each issue classification method the created SDP dataset is 80/20 split, with 80% of the dataset being used for training and 20% for testing. This is done separately for each repetition. The training set uses the labels derived from the issue classification approach in focus, while the test set uses labels derived from manual issue labeling. For each model and each SDP dataset we present the obtained Precision, Recall and MCC score. Matthews's correlation coefficient (MCC) is a metric describing the correlation between real and predicted values. It ranges from -1 to 1 with -1 representing a completely faulty prediction, 0 representing a completely random prediction and 1 representing a perfect prediction. The MCC score is used to validate the performance of the models and make sure that they are performing better than random guessing baselines. To summarize, F1 scores refer to issue classification results and MCC scores refer to SDP results.

For each method, we mention the minimum, average and maximum false positive and false negative share. For each method and for each repository, the percentage of false positives is calculated by dividing the false positive count with the total instance count. Similarly, the false negative percentage is calculated by dividing the false negative count with the total instance count. The minimum false positive percentage for a method is the minimal value obtained across different repositories. The maximum false positive percentage for a method is the maximum value obtained across different repositories. The average is the sum of all obtained values divided by the number of repositories. Analogously, things are calculated for false negatives.

We trained Logistic Regression (LR), Decision Trees (DTC), Naive Bayes (NB) and K-Nearest Neighbours (KNN) models on the SDP datasets derived from manually labeled issues. Please note that the instances of these datasets are created from source code files so in this instance we are performing file level SDP.

The results of issue classification for each classification method are summarized in Table 9. The table displays the Precision, Recall and F1 score of each issue classification

model on each repository. Further, the impact this has on the resulting SDP dataset is shown in Table 10 by listing the number of True Positives, True Negatives, False Positives and False Negatives induced in the resulting SDP dataset. The ground truth from which TP, TN, FP and FN are calculated comes from the manually labeled issue classes. Finally, the performances of the models trained on the derived SDP dataset are shown in Table 11. For each SDP model, on each dataset created from a specific repository using a specific issue classification method to derive the SDP instance labels the table displays the Precision, Recall and MCC score of the that model.

For the sake of readability, the rest of this section is divided into five subsections. The first subsection presents the results obtained using KWM. The second subsection presents the results obtained using the IKWM approach. The third subsection presents the results obtained using the FastText model. The fourth subsection presents the results obtained using the RoBERTa model. Finally, the last subsection summarizes the obtained results.

A. SIMPLE KEYWORD MATCHING

Since this is an unsupervised machine learning approach, the whole issue classification dataset can be used for evaluation. We search for keywords in the issue description and in the issue title. By analyzing the results reported in Table 10 we can observe that the SDP dataset created using the KWM method consists of 13.2093% up to 58.3333% of false positives and 0.0000% up to 3.4335% of false negatives. On average there are 38.4181% of false positives and 1.7780% of false negatives. From this we see that the KWM method is prone to false positives.

B. IMPROVED KEYWORD MATCHING

For each repository we perform issue classification with the IKWM method. By analyzing the results reported in Table 10 we can observe that the SDP dataset created using the IKWM method consists of 14.9261% up to 46.0838% of false positives and 0.0000% up to 5.1979% of false negatives. On average there are 30.7992% of false positives and 1.9316% of false negatives. When comparing these results to those obtained using the KWM approach, on average, we see a drop of false positives from 38.4181% to 30.7992% and a slight increase of false negatives from 1.7780% to 1.9316% of the resulting SDP dataset.

C. FastText

For each repository we perform issue classification with the FastText model. By analyzing the results reported in Table 10 we can observe that the SDP dataset created using the FastText model consists of 0.0000% up to 19.4444% of false positives and 6.0109% up to 18.4615% of false negatives. On average there are 7.4104% of false positives and 9.7246% of false negatives. When comparing these results to those of previous methods we see a significant drop in the number of false positives at the expanse of false negatives. However, the overall amount of noise in the dataset is reduced. With the IKWM method, on average 32.7308% of the SDP dataset was mislabeled while using this approach that number has been reduced to 17.1350%.

D. RoBERTa

For each repository we perform issue classification with the RoBERTa model. By analyzing the results reported in Table 10 we can observe that the SDP dataset created using the RoBERTa model consists of 0.5464% up to 23.6111% of false positives and 0.5875% up to 11.8598% of false negatives. On average there are 9.3369% of false positives and 5.0272% of false negatives.

When comparing these results to the performance of KWM and IKWM we see a significant drop in the number of false positives and an increase in the number of false negatives. When comparing them to the performance of FastText there is a slight increase in the number of false positives, but a noticeable drop in the number of false negatives. Overall, on average, the number of mislabeled instances has been reduced to 14.3641%.

If we analyze the model performances reported in Table 11 we see that in most cases models achieve superior performance on SDP datasets created using RoBERTa when compared to other methods. Also, from the positive MCC values we can easily determine that the model is not performing random guessing.

To statistically validate the achieved results, we compare MCC score distributions achieved by models trained on RoBERTa SDP datasets and models trained on datasets created using other methods. When comparing two samples we first test if both come from normal distributions (with $p = 0.05$). The null hypothesis of the normality test is that the sample comes from the normal distribution. If we fail to reject the null hypothesis for both distributions, we assume that they come from normal distributions and compare them using a Student's T Test [93] (with $p = 0.05$). If the normality test null hypothesis is rejected for at least one distribution, we apply a Mann-Whitney U Test [94] (with $p = 0.05$). Student's T Test determines if two samples drawn from normal distributions have the same expected value. The null hypothesis states that the distributions from which the samples are drawn have the same expected value. If it is rejected, the sample distributions are different with statistical significance. Mann-Whitney U Test is a non-parametric statistical significance test which determines if the two samples come from different distributions. By using a non-parametric test, we are not assuming any specific distributions. The null hypothesis states that there is no difference between the distributions from which the samples are drawn. If it is rejected, the sample distributions are different with statistical significance.

For each repository, there are 3 additional issue classification methods and 4 SDP models are trained, meaning that for each repository there are 12 configurations we are comparing to. Given that we have 7 repositories, that results in 84 configuration where we are comparing the performance of

SDP models trained on SDP datasets created using RoBERTa to other methods.

Models trained on SDP datasets created using RoBERTa had superior performance compared to other methods 65 out of 84 times, and statistically significant superior performance 55 out of 84 times, which is equal to 65.4761% of the time.

We analyzed the model performances reported in Table 11 to see how models trained on datasets created using RoBERTa compare to those trained on golden datasets.

Again, we used the same statistical procedure with the same p values to see if there is a difference in distribution performance. Out of 28 configurations, in 17 cases the test failed to reject the null hypothesis meaning it could not differ between the performance distributions with a statistical significance.

We used t-SNE to visualize the classification of issues by the best-performing model for each repository. The red dots represent defect related issues and the blue dots represent non-defect related issues. The visualization demonstrates how the models have learned to differentiate between defective and non-defective issues. The visualizations are shown in Figure 7.

E. RESULT SUMMARY

All obtained results are presented in Table 9, Table 10 and Table 11. Table 9 presents the issue classification results. Table 10 presents the influence of issue classification on SDP dataset quality. Finally, Table 11 presents the SDP performance of models trained on derived SDP datasets.

To briefly summarize, when applying a KWM to issue classification the resulting SDP datasets had, on average, 38.4181% of false positives and 1.7780% of false negatives, meaning a total of 40.1961% of the dataset was mislabeled. The situation improved when IKWM was applied. With IKWM the resulting datasets, on average contained 30.7992% of false positives and 1.9316% of false negatives, meaning a total of 32.7308% of the dataset was mislabeled. SDP datasets created using the FastText model, on average have 7.4104% of false positives and 9.7246% of false negatives, meaning a total of 17.1350% of the dataset is mislabeled. Finally, datasets created using the RoBERTa model, on average have 9.3369% of false positives and 5.0272% of false negatives, meaning a total of 14.3641% of the dataset is mislabeled.

Kim et al. [10] put an acceptable limit on 20% of FP and 20% of FN. They state that beyond that level there is severe degradation in model performance. Pandey and Tripathi [16] were stricter and put an acceptable limit on 10% of the dataset consisting of incorrectly labeled instances. They state that after that point there is severe performance degradation.

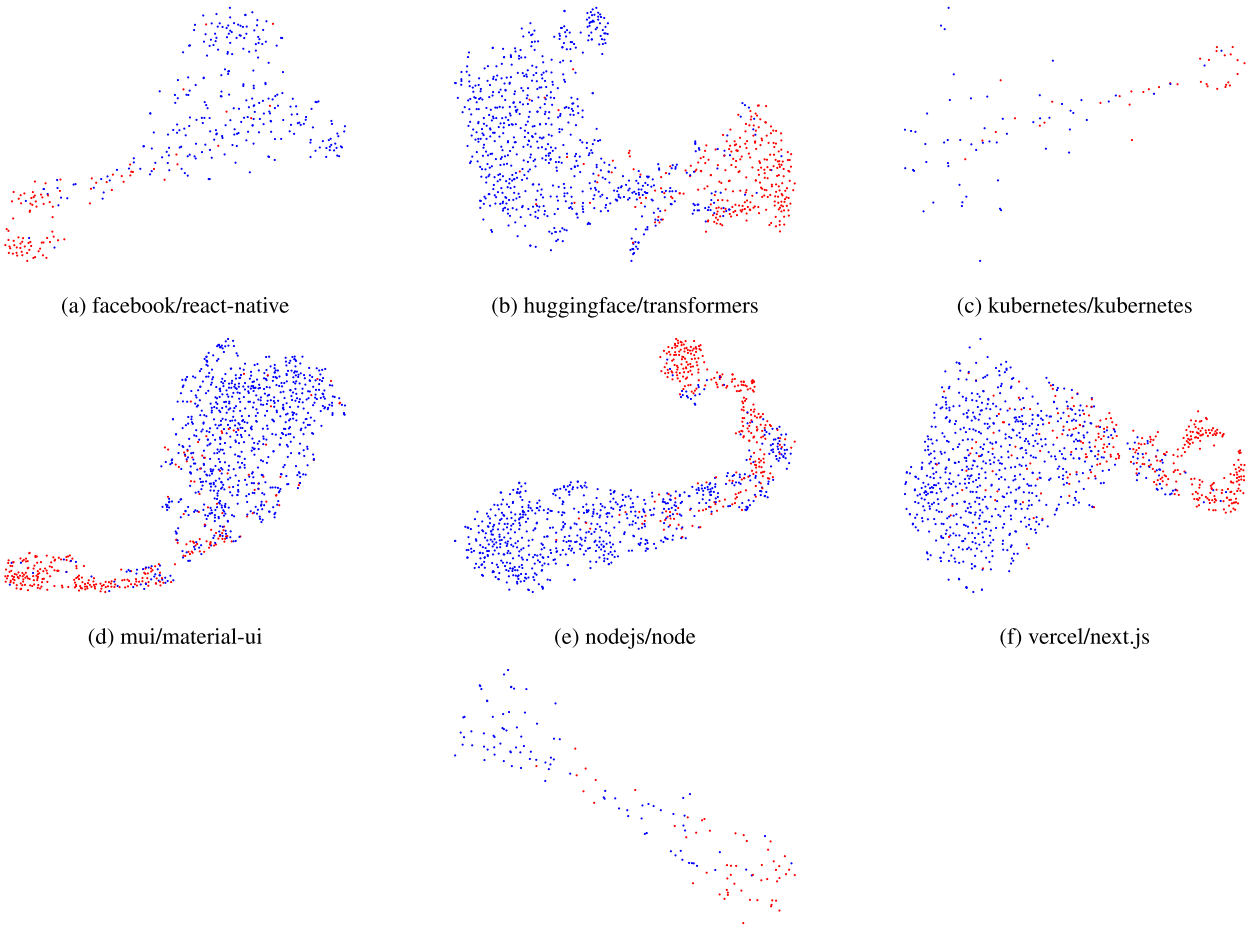
We see that the KWM approach results in datasets which have a high number of false positives and are beyond the limits specified by previous researchers. IKWM improves the dataset quality, but not enough to specify the laid-out

TABLE 9. Issue classification results.

Repository	Model	Precision	Recall	F1
react-native	KWM	0.5600	0.7527	0.6422
react-native	IKWM	0.5600	0.7527	0.6422
react-native	FastText	0.5200	0.8387	0.6420
react-native	RoBERTa	0.8795	0.7850	**0.8296**
transformers	KWM	0.4080	0.9173	0.5648
transformers	IKWM	0.4188	0.9460	0.5806
transformers	FastText	0.5288	0.9245	0.6728
transformers	RoBERTa	0.7278	0.9425	**0.8213**
kubernetes	KWM	0.4853	0.9167	0.6346
kubernetes	IKWM	0.7143	0.8333	0.7692
kubernetes	FastText	0.6482	0.9722	**0.7778**
kubernetes	RoBERTa	0.8696	0.5556	0.6780
material-ui	KWM	0.4965	0.7095	0.5842
material-ui	IKWM	0.4942	0.7162	0.5848
material-ui	FastText	0.6201	0.6892	0.6528
material-ui	RoBERTa	0.7467	0.7669	**0.7567**
node	KWM	0.4517	0.5748	0.5059
node	IKWM	0.4695	0.8439	0.6033
node	FastText	0.4745	0.8638	0.6125
node	RoBERTa	0.6434	0.8870	**0.7458**
next.js	KWM	0.6559	0.8018	0.7215
next.js	IKWM	0.6559	0.8018	0.7215
next.js	FastText	0.5351	0.8842	0.6667
next.js	RoBERTa	0.8577	0.6250	**0.7231**
youtube-dl	KWM	0.3939	0.9455	0.5562
youtube-dl	IKWM	0.5177	0.8000	0.6286
youtube-dl	FastText	0.5313	0.9273	0.6755
youtube-dl	RoBERTa	0.6076	0.8727	**0.7164**

TABLE 10. Issue classification influence on SDP.

Repository	Model	TP	TN	FP	FN
react-native	KWM	100	238	197	14
react-native	IKWM	100	238	197	14
react-native	FastText	112	182	253	2
react-native	RoBERTa	100	432	**3**	**14**
transformers	KWM	308	110	429	4
transformers	IKWM	310	107	432	2
transformers	FastText	309	389	150	3
transformers	RoBERTa	307	470	**69**	**5**
kubernetes	KWM	91	29	75	0
kubernetes	IKWM	48	69	35	43
kubernetes	FastText	91	58	46	0
kubernetes	RoBERTa	73	94	**10**	**18**
material-ui	KWM	1118	630	277	72
material-ui	IKWM	1118	626	281	72
material-ui	FastText	1081	594	313	109
material-ui	RoBERTa	1082	829	**78**	**108**
node	KWM	427	571	1163	27
node	IKWM	435	516	1218	19
node	FastText	397	774	960	57
node	RoBERTa	418	1320	**414**	**36**
next.js	KWM	467	678	289	50
next.js	IKWM	467	678	289	50
next.js	FastText	484	451	516	33
next.js	RoBERTa	341	888	**79**	**176**
youtube-dl	KWM	23	6	42	1
youtube-dl	IKWM	21	24	24	3
youtube-dl	FastText	22	23	25	2
youtube-dl	RoBERTa	21	31	**17**	**3**



(a) facebook/react-native (b) huggingface/transformers (c) kubernetes/kubernetes

(d) mui/material-ui (e) nodejs/node (f) vercel/next.js

(g) ytdl-org/youtube-dl

FIGURE 7. tSNE of RoBERTa embeddings.

criteria. The FastText model reduces the number of false positives and false negatives to a quantity acceptable by the criteria proposed by Kim, but not the one proposed by Pandey. RoBERTa further reduces the amount of noise, but still fails to meet the criteria proposed by Pandey.

We investigated the impact on model performance and found that RoBERTa produces superior, statistically relevant performance in 55 out of 84 times and failed to show a statistically relevant performance difference between models trained on RoBERTa derived SDP datasets and golden datasts 17 out of 28 times.

VII. THREATS TO VALIDITY

This is an empirical study and as such has its own threats to validity. We have identified four possible threats:

Manual issue classification was performed as part of this study. To reduce the impact of this threat multiple people labeled the issues and a majority vote was then taken. However, since the authors do not possess in-depth knowledge of all the selected repositories, incorrect classification might have still occurred and influenced the results. This is one of the reasons we have made the created dataset publicly available where it can be subject to independent assessment.

Open-source software repositories were used as a data source for this study. However, they might not be representative of all software repositories and thus they might introduce a bias into the reported results. One of the ways we have tried to alleviate this problem is by sampling multiple repositories.

English issues were the only ones used in this study. This might introduce a language-based bias. Further studies could be conducted to investigate this issue.

SDP dataset construction methodology used in this study is not the only possible option. For instance, if JIT-SDP was considered then the dataset construction process would be different as it looks into the commit being bug inducing or not. Even if different features were used the effect might be different. One way we have tried to minimize this impact is by using a mixture of code complexity, process, and semantic features. However, the fact remains that a different process might obtain different results. Further studies could be conducted to investigate this effect on different dataset construction methodologies.

TABLE 11. SDP results.

Repository	Model	Dataset Source	Precision	Recall	MCC
react-native	DTC	Manual	0.5763	0.6020	*0.4723*
react-native	DTC	KWM	0.2601	0.6729	0.1271
react-native	DTC	IKWM	0.2337	0.6455	0.1076
react-native	DTC	FastText	0.2627	0.8026	0.1589
react-native	DTC	RoBERTa	0.6089	0.5028	**0.4460**
react-native	NB	Manual	0.3499	0.2622	*0.1427*
react-native	NB	KWM	0.2741	0.3686	0.0400
react-native	NB	IKWM	0.2395	0.3547	0.0020
react-native	NB	FastText	0.4385	0.4022	**0.2603**
react-native	NB	RoBERTa	0.4439	0.2275	0.1973
react-native	KNN	Manual	0.5103	0.2395	*0.2378*
react-native	KNN	KWM	0.2430	0.5987	0.0791
react-native	KNN	IKWM	0.2097	0.5802	0.0444
react-native	KNN	FastText	0.2602	0.8045	0.1534
react-native	KNN	RoBERTa	0.6071	0.2038	**0.2614**
react-native	LR	Manual	0.7321	0.4586	*0.4947*
react-native	LR	KWM	0.2810	0.7119	0.1794
react-native	LR	IKWM	0.2584	0.7079	0.1732
react-native	LR	FastText	0.2838	0.8582	0.2256
react-native	LR	RoBERTa	0.7464	0.3750	**0.4489**
transformers	DTC	Manual	0.5914	0.5918	*0.3548*
transformers	DTC	KWM	0.4006	0.9185	0.1517
transformers	DTC	IKWM	0.3989	0.9095	0.1410
transformers	DTC	FastText	0.5520	0.7879	**0.3914**
transformers	DTC	RoBERTa	0.5630	0.6790	0.3655
transformers	NB	Manual	0.4367	0.8260	*0.2193*
transformers	NB	KWM	0.5306	0.4047	0.2110
transformers	NB	IKWM	0.5154	0.4223	0.1994
transformers	NB	FastText	0.4953	0.8196	0.3119
transformers	NB	RoBERTa	0.4189	0.8375	0.1841
transformers	KNN	Manual	0.6478	0.4600	*0.3454*
transformers	KNN	KWM	0.3864	0.9868	0.1470
transformers	KNN	IKWM	0.3960	0.9343	0.1459
transformers	KNN	FastText	0.5039	0.6977	0.2763
transformers	KNN	RoBERTa	0.5608	0.6121	**0.3298**
transformers	LR	Manual	0.7203	0.5551	*0.4594*
transformers	LR	KWM	0.3776	0.9786	0.0848
transformers	LR	IKWM	0.3779	0.9831	0.0888
transformers	LR	FastText	0.5718	0.7802	0.4138
transformers	LR	RoBERTa	0.6328	0.6723	**0.4422**
kubernetes	DTC	Manual	0.6642	0.6596	*0.3924*
kubernetes	DTC	KWM	0.4780	0.8868	0.1110
kubernetes	DTC	IKWM	0.4576	0.4343	-0.0168
kubernetes	DTC	FastText	0.5404	0.7725	0.1919
kubernetes	DTC	RoBERTa	0.5912	0.5702	**0.2692**
kubernetes	NB	Manual	0.6274	0.0910	*0.1445*
kubernetes	NB	KWM	0.3975	0.5053	-0.1280
kubernetes	NB	IKWM	0.4378	0.1175	0.0317
kubernetes	NB	FastText	0.6615	0.1571	**0.1156**
kubernetes	NB	RoBERTa	0.4937	0.0640	0.1035
kubernetes	KNN	Manual	0.5490	0.5585	*0.1901*
kubernetes	KNN	KWM	0.4566	0.9554	0.0392
kubernetes	KNN	IKWM	0.4578	0.3423	-0.0121
kubernetes	KNN	FastText	0.5416	0.7626	0.1913
kubernetes	KNN	RoBERTa	0.5384	0.4889	0.1631
kubernetes	LR	Manual	0.7220	0.7640	*0.5288*
kubernetes	LR	KWM	0.4764	0.9927	0.1851
kubernetes	LR	IKWM	0.4368	0.3088	-0.0453
kubernetes	LR	FastText	0.5578	0.9057	0.3059
kubernetes	LR	RoBERTa	0.7354	0.6480	**0.4755**
material-ui	DTC	Manual	0.8198	0.8288	*0.5889*
material-ui	DTC	KWM	0.7215	0.8460	0.4426
material-ui	DTC	IKWM	0.7264	0.8539	0.4521
material-ui	DTC	FastText	0.7041	0.8145	0.3910
material-ui	DTC	RoBERTa	0.8089	0.7906	**0.5512**
material-ui	NB	Manual	0.7649	0.3652	*0.2460*
material-ui	NB	KWM	0.7154	0.2831	0.1631
material-ui	NB	IKWM	0.7135	0.2838	0.1572
material-ui	NB	FastText	0.7074	0.2679	0.1526
material-ui	NB	RoBERTa	0.7465	0.3224	**0.2135**
material-ui	KNN	Manual	0.8524	0.8329	*0.6405*
material-ui	KNN	KWM	0.7446	0.8674	0.5037
material-ui	KNN	IKWM	0.7318	0.8390	0.4506
material-ui	KNN	FastText	0.7145	0.8508	0.4379
material-ui	KNN	RoBERTa	0.8496	0.7920	**0.6091**
material-ui	LR	Manual	0.8337	0.8160	*0.5985*
material-ui	LR	KWM	0.7279	0.8890	0.4913
material-ui	LR	IKWM	0.7278	0.8878	0.4836
material-ui	LR	FastText	0.7041	0.8682	0.4310
material-ui	LR	RoBERTa	0.8341	0.8080	**0.6010**
node	DTC	Manual	0.6252	0.6288	*0.5281*
node	DTC	KWM	0.2569	0.8887	0.1886
node	DTC	IKWM	0.2502	0.9066	0.1838
node	DTC	FastText	0.2830	0.8312	0.2302
node	DTC	RoBERTa	0.4212	0.7515	**0.3985**
node	NB	Manual	0.4350	0.3214	*0.2381*
node	NB	KWM	0.3177	0.3287	0.1403
node	NB	IKWM	0.3170	0.3272	0.1340
node	NB	FastText	0.2979	0.2493	0.1012
node	NB	RoBERTa	0.3595	0.2528	**0.1508**
node	KNN	Manual	0.6198	0.2572	*0.3103*
node	KNN	KWM	0.2275	0.8750	0.0894
node	KNN	IKWM	0.2356	0.8378	0.1166
node	KNN	FastText	0.2585	0.7597	0.1547
node	KNN	RoBERTa	0.3732	0.6294	**0.2940**
node	LR	Manual	0.7836	0.6124	*0.6239*
node	LR	KWM	0.2491	0.9577	0.1957
node	LR	IKWM	0.2372	0.9774	0.1771
node	LR	FastText	0.2822	0.9315	0.2692
node	LR	RoBERTa	0.5044	0.8051	**0.5113**
next.js	DTC	Manual	0.5088	0.5184	*0.2493*
next.js	DTC	KWM	0.4540	0.6394	0.2128
next.js	DTC	IKWM	0.4406	0.6523	0.2250
next.js	DTC	FastText	0.3965	0.7487	0.1434
next.js	DTC	RoBERTa	0.5312	0.4608	**0.2548**
next.js	NB	Manual	0.5466	0.2492	*0.1799*
next.js	NB	KWM	0.5335	0.2471	0.1681
next.js	NB	IKWM	0.5044	0.2417	0.1581
next.js	NB	FastText	0.5051	0.2760	0.1601
next.js	NB	RoBERTa	0.4725	0.2496	0.1241
next.js	KNN	Manual	0.5662	0.3825	*0.2537*
next.js	KNN	KWM	0.4692	0.5970	**0.2222**
next.js	KNN	IKWM	0.4263	0.6143	0.1901
next.js	KNN	FastText	0.3885	0.7173	0.1159
next.js	KNN	RoBERTa	0.5237	0.3870	0.2172
next.js	LR	Manual	0.6838	0.4059	*0.3588*
next.js	LR	KWM	0.4995	0.6074	0.2675
next.js	LR	IKWM	0.4947	0.6313	0.2947
next.js	LR	FastText	0.3880	0.8951	0.1710
next.js	LR	RoBERTa	0.7242	0.3040	**0.3259**
youtube-dl	DTC	Manual	0.5965	0.5634	*0.3843*
youtube-dl	DTC	KWM	0.3191	0.8577	-0.0544
youtube-dl	DTC	IKWM	0.3911	0.6868	0.0938
youtube-dl	DTC	FastText	0.3047	0.6308	-0.0118
youtube-dl	DTC	RoBERTa	0.4089	0.6491	**0.1668**
youtube-dl	NB	Manual	0.7002	0.6763	*0.5471*
youtube-dl	NB	KWM	0.6457	0.8594	0.5890
youtube-dl	NB	IKWM	0.7746	0.7180	**0.6254**
youtube-dl	NB	FastText	0.6760	0.7227	0.5438
youtube-dl	NB	RoBERTa	0.6718	0.6416	0.4774
youtube-dl	KNN	Manual	0.4589	0.2176	*0.1651*
youtube-dl	KNN	KWM	0.3283	0.9889	-0.0218
youtube-dl	KNN	IKWM	0.3750	0.6847	0.0695
youtube-dl	KNN	FastText	0.3095	0.6487	-0.0106
youtube-dl	KNN	RoBERTa	0.4520	0.6784	**0.2366**
youtube-dl	LR	Manual	0.9200	0.8024	*0.7923*
youtube-dl	LR	KWM	0.3194	0.9514	-0.0924
youtube-dl	LR	IKWM	0.4951	0.9739	0.4390
youtube-dl	LR	FastText	0.4151	0.9627	0.3420
youtube-dl	LR	RoBERTa	0.6104	0.8648	**0.5410**

VIII. CONCLUSION

As part of this study, we investigate the impact of issue classification on SDP dataset quality and resulting model performance. In order to do this, we created new datasets by mining 7 popular open-source repositories. For every repository, we collected all commit data and all issue data. By analyzing commit messages, we identified *issues-of-interest*. These are issues referenced by at least one commit. Commits containing at least one issue reference are considered *commits-of-interest*. We then identified source code files edited exclusively by *commits-of-interest* and call them *file-of-interest*. For each repository, we sampled at least 1000 IOI, manually inspected and labeled them. We determined which FOI are related to the labeled issues. From them, using code complexity analysis, process analysis and GraphCodeBERT we created SDP dataset instances. The golden labels of these instances are derived from the manually labeled issues. We then investigated how using different methods for issue classification would influence the created SDP datasets and performance of standard models trained on these datasets. Issue classification was done using a keyword matching heuristic, an improved keyword matching heuristic, a FastText model and a RoBERTa model. For each resulting SDP dataset we trained Logistic Regression, Decision Trees, Naive Bayes and K-Nearest Neighbours models.

From the achieved results we see that applying KWM to issue classification produces SDP datasets with an average 38.4181% of false positives and 1.7780% of false negatives, so a total of 40.1961% of mislabeled instances. IKWM produces datasets with an average 30.7992% of false positives and 1.9316% of false negatives, so a total of 32.7308% of mislabeled instances. FastText produced datasets have 7.4104% of false positives and 9.7246% of false negatives, so a total of 17.1350% of the dataset is mislabeled. Finally, datasets created using the RoBERTa model contain an average 9.3369% of false positives and 5.0272% of false negatives, totaling 14.3641% of mislabeled instances. The obtained results clearly show that of the inspected issue classification approaches RoBERTa produces the highest quality SDP datasets.

We then investigated the impact this has on model performance and found that models trained on RoBERTa derived SDP datasets outperformed counterparts trained on differently derived SDP datasets 65 out of 84 times, 55 of which were statistically relevant. When comparing their performance to those trained on golden datasets we could not show a statistically relevant performance difference 17 out of 28 times.

Based on the presented results we advocate that the research community use advanced NLP models such as RoBERTa when creating datasets for software defect prediction if issue classes cannot be determined with certainty. In our public repository we provide pre-trained models for issue classification.

To support further scientific inquiry in this research area, and put our own work under scrutiny, we have made all our source code, labeling application, created datasets and models publicly available. They can be found on GitHub.[2]

REFERENCES

[1] H. Krasner, "The cost of poor software quality in the U.S.: A 2020 report," in *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, 2021, pp. 1–46.
[2] M. K. Thota, F. H. Shajin, and P. Rajesh, "Survey on software defect prediction techniques," *Int. J. Appl. Sci. Eng.*, vol. 17, no. 4, pp. 331–344, 2020.
[3] A. Hasanpour, P. Farzi, A. Tehrani, and R. Akbari, "Software defect prediction based on deep learning models: Performance study," 2020, *arXiv:2004.02589*.
[4] Z. Li, X.-Y. Jing, and X. Zhu, "Progress on approaches to software defect prediction," *IET Softw.*, vol. 12, no. 3, pp. 161–175, Jun. 2018.
[5] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced? Bias in bug-fix datasets," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, Aug. 2009, pp. 121–130.
[6] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu, "Sample size vs. bias in defect prediction," in *Proc. 9th Joint Meeting Found. Softw. Eng.*, Aug. 2013, pp. 147–157.
[7] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *Proc. Int. Conf. Softw. Maintenance*, 2000, pp. 120–130.
[8] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, 2005.
[9] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 392–401.
[10] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Proc. 33rd Int. Conf. Softw. Eng.*, May 2011, pp. 481–490.
[11] J. Wang, X. Zhang, and L. Chen, "How well do pre-trained contextual language representations recommend labels for GitHub issues?" *Knowl.-Based Syst.*, vol. 232, Nov. 2021, Art. no. 107476, doi: 10.1016/j.knosys.2021.107476.
[12] S. Herbold, A. Trautsch, and F. Trautsch, "On the feasibility of automated issue type prediction," 2020, *arXiv:2003.05357*.
[13] R. Kallis, A. Di Sorbo, G. Canfora, and S. Panichella, "Predicting issue types on GitHub," *Sci. Comput. Program.*, vol. 205, May 2021, Art. no. 102598, doi: 10.1016/j.scico.2020.102598.
[14] M. Siddiq and J. S. Santos, "BERT-based GitHub issue report classification," in *Proc. SPIEIEEE/ACM 1st Int. Workshop Natural Lang.-Based Softw. Eng. (NLBSE)*, May 2022, pp. 33–36.
[15] C. Seiffert, T. M. Khoshgoftaar, J. Van Hulse, and A. Folleco, "An empirical study of the classification performance of learners on imbalanced and noisy software quality data," *Inf. Sci.*, vol. 259, pp. 571–595, Feb. 2014, doi: 10.1016/j.ins.2010.12.016.
[16] S. K. Pandey and A. K. Tripathi, "An empirical study toward dealing with noise and class imbalance issues in software defect prediction," *Soft Comput.*, vol. 25, no. 21, pp. 13465–13492, Nov. 2021, doi: 10.1007/s00500-021-06096-3.
[17] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto, "The impact of mislabelling on the performance and interpretation of defect prediction models," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, May 2015, pp. 812–823, doi: 10.1109/ICSE.2015.93.
[18] M. A. Azmain, S. S. Khan, N. T. Niloy, and A. Kabir, "Impact of label noise and efficacy of noise filters in software defect prediction," in *Proc. 32nd Int. Conf. Softw. Eng. Knowl. Eng.*, 2020, pp. 347–352, doi: 10.6084/m9.figshare.14191400.v1.
[19] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement? A text-based approach to classify change requests," in *Proc. Conf. Center Adv. Stud. Collaborative Res., Meeting Minds*, 2008, pp. 304–318.
[20] S. Zhang, S. Jiang, and Y. Yan, "A software defect prediction approach based on BiGAN anomaly detection," *Sci. Program.*, vol. 2022, pp. 1–13, Apr. 2022.
[21] P. Afric, L. Sikic, A. S. Kurdija, and M. Silic, "REPD: Source code defect prediction as anomaly detection," *J. Syst. Softw.*, vol. 168, Oct. 2020, Art. no. 110641.

[2]https://github.com/pa1511/Empirical-Study-How-Issue-Classification-Influences-Software-Defect-Prediction

[22] S. Kim, T. Zimmermann, E. J. Whitehead, and A. Zeller, "Predicting faults from cached history," in *Proc. 1st India Softw. Eng. Conf.*, Feb. 2008, pp. 489–498.

[23] N. Li, M. Shepperd, and Y. Guo, "A systematic review of unsupervised learning techniques for software defect prediction," 2019, *arXiv:1907.12027*.

[24] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. 38th Int. Conf. Softw. Eng.*, May 2016, pp. 297–308, doi: 10.1145/2884781.2884804.

[25] X. Huo, Y. Yang, M. Li, and D.-C. Zhan, "Learning semantic features for software defect prediction by code comments embedding," in *Proc. IEEE Int. Conf. Data Mining (ICDM)*, Nov. 2018, pp. 1049–1054, doi: 10.1109/ICDM.2018.00133.

[26] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 46, no. 12, pp. 1267–1293, Dec. 2020, doi: 10.1109/TSE.2018.2877612.

[27] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "GraphCodeBERT: Pre-training code representations with data flow," 2020, *arXiv:2009.08366*.

[28] F. Akiyama, "An example of software system debugging," in *Proc. IFIP Congr.*, vol. 1, C. V. Freiman, J. E. Griffith, and J. L. Rosenfeld, Eds. North-Holland, 1971, pp. 353–359. [Online]. Available: http://dblp.uni-trier.de/db/conf/ifip/ifip71-1.html#Akiyama71 and https://www.bibsonomy.org/bibtex/263b966654f26898e4d67b93d0ed640b9/dblp

[29] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "ReLink: Recovering links between bugs and changes," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.*, Sep. 2011, pp. 15–25.

[30] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Multi-layered approach for recovering links between bug reports and fixes," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, Nov. 2012, doi: 10.1145/2393596.2393671.

[31] R. Ferenc, Z. Tóth, G. Ladányi, I. Siket, and T. Gyimóthy, "A public unified bug dataset for Java," in *Proc. 14th Int. Conf. Predictive Models Data Anal. Softw. Eng.*, Oct. 2018, pp. 12–21.

[32] 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," *Softw. Quality J.*, vol. 28, no. 4, pp. 1447–1506, Dec. 2020.

[33] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proc. 3rd Int. Workshop Predictor Models Softw. Eng.*, May 2007, pp. 1–9.

[34] S. J. Sayyad and T. J. Menzies, "The PROMISE repository of software engineering databases," School Inf. Technol. Eng., Univ. Ottawa, Ottawa, ON, Canada, 2005, [Online]. Available: http://promise.site.uottawa.ca/SERepository

[35] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, Jan. 2007, doi: 10.1109/TSE.2007.256941.

[36] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data quality: Some comments on the NASA software defect datasets," *IEEE Trans. Softw. Eng.*, vol. 39, no. 9, pp. 1208–1215, Sep. 2013, doi: 10.1109/TSE.2013.11.

[37] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson, "The misuse of the NASA metrics data program data sets for automated software defect prediction," in *Proc. 15th Annu. Conf. Eval. Assessment Softw. Eng.*, 2011, pp. 96–103, doi: 10.1049/ic.2011.0012.

[38] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976, doi: 10.1109/TSE.1976.233837.

[39] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. Amsterdam, The Netherlands: Elsevier, 1977.

[40] M. Hitz and B. Montazeri, "Chidamber and Kemerer's metrics suite: A measurement theory perspective," *IEEE Trans. Softw. Eng.*, vol. 22, no. 4, pp. 267–271, Apr. 1996, doi: 10.1109/32.491650.

[41] S. Matsumoto, Y. Kamei, A. Monden, K.-I. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in *Proc. 6th Int. Conf. Predictive Models Softw. Eng.*, Sep. 2010, pp. 1–9, doi: 10.1145/1868328.1868356.

[42] J. Byun, S. Rhew, M. Hwang, V. Sugumara, S. Park, and S. Park, "Metrics for measuring the consistencies of requirements with objectives and constraints," *Requirements Eng.*, vol. 19, no. 1, pp. 89–104, Mar. 2014.

[43] E. Mnkandla and B. Mpofu, "Software defect prediction using process metrics elasticsearch engine case study," in *Proc. Int. Conf. Adv. Comput. Commun. Eng. (ICACCE)*, Nov. 2016, pp. 254–260, doi: 10.1109/ICACCE.2016.8073757.

[44] M. Jureczko and L. Madeyski, "A review of process metrics in defect prediction studies," *Metody Informatyki Stosowanej*, vol. 5, pp. 133–145, May 2011.

[45] S. Ramadhina, R. B. Bahawares, I. Hermadi, A. I. Suroso, A. Rodoni, and Y. Arkeman, "Software defect prediction using process metrics systematic literature review: Dataset and granularity level," in *Proc. 9th Int. Conf. Cyber IT Service Manage. (CITSM)*, Sep. 2021, pp. 1–7, doi: 10.1109/CITSM52892.2021.9587932.

[46] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change bursts as defect predictors," in *Proc. IEEE 21st Int. Symp. Softw. Rel. Eng.*, Nov. 2010, pp. 309–318, doi: 10.1109/ISSRE.2010.25.

[47] G. R. Choudhary, S. Kumar, K. Kumar, A. Mishra, and C. Catal, "Empirical analysis of change metrics for software fault prediction," *Comput. Electr. Eng.*, vol. 67, pp. 15–24, Apr. 2018.

[48] W. Rhmann, B. Pandey, G. Ansari, and D. K. Pandey, "Software fault prediction based on change metrics using hybrid algorithms: An empirical study," *J. King Saud Univ.-Comput. Inf. Sci.*, vol. 32, no. 4, pp. 419–424, May 2020, doi: 10.1016/j.jksuci.2019.03.006.

[49] L. Sikic, P. Afric, A. S. Kurdija, and M. Silic, "Improving software defect prediction by aggregated change metrics," *IEEE Access*, vol. 9, pp. 19391–19411, 2021, doi: 10.1109/ACCESS.2021.3054948.

[50] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data vs. domain vs. process," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, Aug. 2009, pp. 91–100.

[51] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, "Heterogeneous defect prediction," *IEEE Trans. Softw. Eng.*, vol. 44, no. 9, pp. 874–896, Sep. 2018.

[52] X. Chen, Y. Mu, K. Liu, Z. Cui, and C. Ni, "Revisiting heterogeneous defect prediction methods: How far are we?" *Inf. Softw. Technol.*, vol. 130, Feb. 2021, Art. no. 106441.

[53] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 10, pp. 1345–1359, Oct. 2010, doi: 10.1109/TKDE.2009.191.

[54] S. J. Pan, I. W. Tsang, J. T. Kwok, and Q. Yang, "Domain adaptation via transfer component analysis," *IEEE Trans. Neural Netw.*, vol. 22, no. 2, pp. 199–210, Feb. 2011, doi: 10.1109/TNN.2010.2091281.

[55] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 382–391, doi: 10.1109/ICSE.2013.6606584.

[56] S. Pal, "Generative adversarial network-based cross-project fault prediction," 2021, *arXiv:2105.07207*.

[57] F. Porto, L. Minku, E. Mendes, and A. Simao, "A systematic study of cross-project defect prediction with meta-learning," 2018, *arXiv:1802.06025*.

[58] S. Watanabe, H. Kaiya, and K. Kaijiri, "Adapting a fault prediction model to allow inter languagereuse," in *Proc. 4th Int. Workshop Predictor Models Softw. Eng.*, May 2008, pp. 19–24, doi: 10.1145/1370788.1370794.

[59] J. Chen, X. Wang, S. Cai, J. Xu, J. Chen, and H. Chen, "A software defect prediction method with metric compensation based on feature selection and transfer learning," *Frontiers Inf. Technol. Electron. Eng.*, vol. 23, no. 5, pp. 715–731, May 2022, doi: 10.1631/FITEE.2100468.

[60] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Softw. Eng.*, vol. 14, no. 5, pp. 540–578, Oct. 2009, doi: 10.1007/s10664-008-9103-7.

[61] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim, "Reducing features to improve bug prediction," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2009, pp. 600–604, doi: 10.1109/ASE.2009.76.

[62] M. Kakkar and S. Jain, "Feature selection in software defect prediction: A comparative study," in *Proc. 6th Int. Conf.-Cloud Syst. Big Data Eng.*, Jan. 2016, pp. 658–663, doi: 10.1109/CONFLUENCE.2016.7508200.

[63] A. Alsaeedi and M. Z. Khan, "Software defect prediction using supervised machine learning and ensemble techniques: A comparative study," *J. Softw. Eng. Appl.*, vol. 12, no. 5, pp. 85–100, 2019, doi: 10.4236/jsea.2019.125007.

[64] F. Matloob, S. Aftab, M. Ahmad, M. A. Khan, A. Fatima, M. Iqbal, W. M. Alruwaili, and N. S. Elmitwally, "Software defect prediction using supervised machine learning techniques: A systematic literature review," *Intell. Automat. Soft Comput.*, vol. 29, no. 2, p. 403, 2021, doi: 10.32604/iasc.2021.017562.

[65] L. Sikic, A. S. Kurdija, K. Vladimir, and M. Silic, "Graph neural network for source code defect prediction," *IEEE Access*, vol. 10, pp. 10402–10415, 2022, doi: 10.1109/ACCESS.2022.3144598.

[66] H. Lu, B. Cukic, and M. Culp, "A semi-supervised approach to software defect prediction," in *Proc. IEEE 38th Annu. Comput. Softw. Appl. Conf.*, Jul. 2014, pp. 416–425, doi: 10.1109/COMPSAC.2014.65.

[67] F. Wu, X.-Y. Jing, X. Dong, J. Cao, M. Xu, H. Zhang, S. Ying, and B. Xu, "Cross-project and within-project semi-supervised software defect prediction problems study using a unified solution," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion (ICSE-C)*, May 2017, pp. 195–197, doi: 10.1109/ICSE-C.2017.72.

[68] G. Calikli, A. Tosun, A. Bener, and M. Celik, "The effect of granularity level on software defect prediction," in *Proc. 24th Int. Symp. Comput. Inf. Sci.*, Sep. 2009, pp. 531–536, doi: 10.1109/ISCIS.2009.5291866.

[69] H. D. Tessema and S. L. Abebe, "Enhancing just-in-time defect prediction using change request-based metrics," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng.*, Mar. 2021, pp. 511–515, doi: 10.1109/SANER50967.2021.00056.

[70] G. Scanniello, C. Gravino, A. Marcus, and T. Menzies, "Class level fault prediction using software clustering," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2013, pp. 640–645, doi: 10.1109/ASE.2013.6693126.

[71] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proc. 28th Int. Conf. Softw. Eng.*, May 2006, pp. 452–461, doi: 10.1145/1134285.1134349.

[72] Y. Khatri and S. K. Singh, "Cross project defect prediction: A comprehensive survey with its SWOT analysis," *Innov. Syst. Softw. Eng.*, vol. 18, no. 2, pp. 263–281, Jun. 2022, doi: 10.1007/s11334-020-00380-5.

[73] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," 2016, *arXiv:1607.01759*.

[74] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, *arXiv:1301.3781*.

[75] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, "A convolutional neural network for modelling sentences," 2014, *arXiv:1404.2188*.

[76] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 28, 2015, pp. 1–15.

[77] D. Shen, Y. Zhang, R. Henao, Q. Su, and L. Carin, "Deconvolutional latent-variable model for text sequence matching," in *Proc. AAAI Conf. Artif. Intell.*, 2018, vol. 32, no. 1, pp. 1–8.

[78] P. Liu, X. Qiu, and X. Huang, "Recurrent neural network for text classification with multi-task learning," 2016, *arXiv:1605.05101*.

[79] M. Seo, S. Min, A. Farhadi, and H. Hajishirzi, "Neural speed reading via skim-RNN," 2017, *arXiv:1711.02085*.

[80] Z. Yang, D. Yang, C. Dyer, X. He, A. J. Smola, and E. H. Hovy, "Hierarchical attention networks for document classification," in *Proc. NAACL*, 2016, pp. 1480–1489, doi: 10.18653/v1/N16-1174.

[81] Z. Lin, M. Feng, C. Nogueira dos Santos, M. Yu, B. Xiang, B. Zhou, and Y. Bengio, "A structured self-attentive sentence embedding," 2017, *arXiv:1703.03130*.

[82] C. Sun, X. Qiu, Y. Xu, and X. Huang, "How to fine-tune BERT for text classification?" 2019, *arXiv:1905.05583*.

[83] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.

[84] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A robustly optimized BERT pretraining approach," 2019, *arXiv:1907.11692*.

[85] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017, *arXiv:1706.03762*.

[86] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2015, pp. 19–27, doi: 10.1109/ICCV.2015.11.

[87] W. L. Taylor, "'Cloze Procedure': A new tool for measuring readability," *Journalism Quart.*, vol. 30, no. 4, pp. 415–433, Sep. 1953, doi: 10.1177/107769905303000401.

[88] D. Cubranic and G. C. Murphy, "Hipikat: Recommending pertinent software development artifacts," in *Proc. 25th Int. Conf. Softw. Eng.*, May 2003, pp. 408–418, doi: 10.1109/ICSE.2003.1201219.

[89] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Proc. Int. Conf. Softw. Maintenance*, 2003, pp. 23–32, doi: 10.1109/ICSM.2003.1235403.

[90] A. Bachmann and A. Bernstein, "Software process data quality and characteristics: A historical view on open and closed source projects," in *Proc. Joint Int. Annu. ERCIM Workshops Princ. Softw. Evol. (IWPSE) Softw. Evol.*, Aug. 2009, pp. 119–128, doi: 10.1145/1595808.1595830.

[91] H. Qing, L. Biwen, S. Beijun, and Y. Xia, "Cross-project software defect prediction using feature-based transfer learning," in *Proc. 7th Asia–Pacific Symp. Internetware*, Nov. 2015, pp. 74–82, doi: 10.1145/2875913.2875944.

[92] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980, doi: 10.1108/eb046814.

[93] D. Kalpić, N. Hlupić, and M. Lovrić, "Student's *t*-tests," in *International Encyclopedia of Statistical Science*, M. Lovric, Ed. Berlin, Germany: Springer, 2011, pp. 1559–1563, doi: 10.1007/978-3-642-04898-2_641.

[94] M. Neuhäuser, "Wilcoxon–Mann–Whitney test," in *International Encyclopedia of Statistical Science*, M. Lovric, Ed. Berlin, Germany: Springer, 2011, pp. 1656–1658, doi: 10.1007/978-3-642-04898-2_615.

PETAR AFRIC was born in Split, Croatia, in 1993. He received the master's degree in computer science from the Faculty of Electrical Engineering and Computing, University of Zagreb, Zagreb, Croatia, in 2018, where he is currently pursuing the Ph.D. degree. He is currently the Chief Technical Officer with the DataBlast, Zagreb. He has previously published in the *Journal of Systems and Software* and presented at the IEEE International Conference on Software Quality, Reliability and Security. His research interests include software defect prediction, software quality assurance, and optimization algorithms.

DAVOR VUKADIN was born in Samobor, Croatia, in 1996. He received the master's degree in computer science from the Faculty of Electrical Engineering and Computing, University of Zagreb, Zagreb, Croatia, in 2020, where he is currently pursuing the Ph.D. degree. He is currently a Researcher with the Faculty of Electrical Engineering and Computing, University of Zagreb. He was published in IEEE Access. His research interests include software defect prediction, natural language processing, and AI (artificial intelligence) explainability.

MARIN SILIC (Member, IEEE) received the Ph.D. degree in computer science from the Faculty of Electrical Engineering and Computing, University of Zagreb, in 2013. He is currently an Associate Professor with the Faculty of Electrical Engineering and Computing, University of Zagreb. He has published several papers in IEEE Transactions on Services Computing, IEEE Transactions on Dependable and Secure Computing, *Journal of Systems and Software*, and *Knowledge-Based Systems*. Also, he has published his research results at the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering and at the IEEE International Conference on Software Quality, Reliability and Security. His research interests include machine learning, data mining, service-oriented computing, and software engineering.

GORAN DELAC (Member, IEEE) received the Ph.D. degree in computer science from the Faculty of Electrical Engineering and Computing, University of Zagreb, in 2014. He is currently an Associate Professor with the Faculty of Electrical Engineering and Computing, University of Zagreb. He has published several papers in IEEE Transactions on Services Computing, IEEE Transactions on Dependable and Secure Computing, and *Knowledge-Based Systems*. Also, he has published his research results at the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering and at the IEEE International Conference on Software Quality, Reliability and Security. His research interests include distributed systems, fault tolerant systems, service-oriented computing, data mining, and machine learning.