Enriching the Semantic Representation of the Source Code with Natural Language-Based Features from Comments for Improving the Performance of Software Defect Prediction

Anamaria Briciu[a], Mihaiela Lupea[b], Gabriela Czibula[c] and Istvan Gergely Czibula[d]
Department of Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania

Keywords: Software Defect Prediction, Machine Learning, Semantic Features, BERT-Based Models, doc2vec, Source Code, Comments.

Abstract: The present study belongs to the new research direction that aims to improve *software defect prediction* by using additional knowledge such as source code comments. The fusion of *programming language* features learned from the code and *natural language* features extracted from the code comments is the proposed semantic representation of a source code. Two types of language models are applied to learn the semantic features: (1) the pre-trained models CodeBERT and RoBERTa for code embedding and textual embedding; (2) doc2vec model used for both, code embedding and comments embedding. These two semantic representations, in two combinations (only code features and code features fused with comment features), are used separately with the XGBoost classifier in the experiments conducted on the Calcite dataset. The results show that the addition of the natural language features from the comments increases the software defect prediction performance.

1 INTRODUCTION

Software defect prediction (SDP) is an important area in software engineering research and practice, as it can improve the software development process by providing automated, early detection of software defects in the system. Besides its importance in measuring a software project evolution, *software defect prediction* assists the process management (Clark and Zubrow, 2001), is useful in predicting software reliability (Zheng, 2009) and in guiding testing and code review (hua Chang et al., 2011). Through all these activities, SDP contributes to a significant decrease of the costs required by the software products' development and maintenance (Hryszko and Madeyski, 2018).

The approaches from the SDP literature are categorized in two types: *within-project* and *cross-project* software defect prediction. Within-project approaches use data from a software project both for training and testing the defect predictor (Zhu et al., 2020), while the cross-project SDP approaches train the SDP model on a set of software systems and then test its performance on different software systems (Jin, 2021).

The importance of employing semantic features from the source code besides software metrics for SDP has been highlighted by (Wang et al., 2016). (Yang et al., 2015) and (Wang et al., 2016) considered Deep Belief Neural Network (DBNN) and source code analysis for predicting software defects. Long Short Term Memory (LSTM) networks were proposed by (Dam et al., 2018) to learn semantic features from the abstract syntax tree (AST) of the code. A Logistic Regression (LR) and a Random Forest classifier were then trained on these semantic features for predicting software defects. (Li et al., 2017) employed a Convolutional Neural Network (CNN) trained on the semantic features learned from the abstract syntax tree combined with the software metrics. A LR classifier applied on the automatically learned features outperformed the DBNN approach introduced by (Wang et al., 2016). (Zhao et al., 2019) applied Siamese parallel fully connected networks to generate new features out of the software metric values, without using the source code.

One of the main difficulties in SDP is the severe

[a] https://orcid.org/0000-0001-6681-7281
[b] https://orcid.org/0000-0002-2117-2018
[c] https://orcid.org/0000-0001-7852-681X
[d] https://orcid.org/0000-0003-0076-584X

Briciu, A., Lupea, M., Czibula, G. and Czibula, I.
Enriching the Semantic Representation of the Source Code with Natural Language-Based Features from Comments for Improving the Performance of Software Defect Prediction.
DOI: 10.5220/0012688400003687

In *Proceedings of the 19th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2024)*, pages 132-143
ISBN: 978-989-758-696-5; ISSN: 2184-4895

imbalance of the data, as the number of software defects in the training data is significantly outweighed by the number of non-defects. Due to this imbalance, the supervised classifiers are often biased to predict the majority class (non-defects). (Zhao et al., 2019) showed that even the deep learning (DL) models are impacted by the imbalance of the defect datasets. Most of the existing SDP approaches based on source code are analysing its AST. Two main limitations of DL models applied on the AST of the source code are that AST tokens do not exploit comments and identifiers which are also relevant in expressing semantic relations (Marcus et al., 2008) and the feature generation process is dependent on the programming language. (hua Chang et al., 2011) highlighted the importance of using quality data for training the defect predictors and the importance of the features used for characterizing the software entities.

From our perspective, a major challenge in the SDP task lies in the selection of features, given the numerous aspects of the source code that a representation must capture to effectively discern between defects and non-defects. Therefore, we emphasize the significance of investigating potentially relevant features that may improve upon existing representations. Code comments and names in the source code may contain relevant information regarding the domain of the problem and may also highlight relations between parts of the code that can not be easily discovered by just analyzing traditional code dependencies (associations, aggregations, inheritance). Many of the defects discovered in a software system are related to non-technical aspects of the system such as: misunderstandings related to the requirements or specification of functions, defects caused by problems in interoperability between different components of the system, mismatches between the semantic meaning of certain variables. These semantic aspects of the system are more likely captured by comments, variable/function/class names in the system than software metrics, AST or other data collected and used by the majority of SDP approaches in the literature.

The present study belongs to the new research in the field of SDP that uses additional knowledge, such as comments, with the aim of obtaining a better semantic representation of the source code. To evaluate the relevance of the new *natural language* (NL) - based features in improving software defect prediction, two SDP approaches are proposed. The first approach combines *programming language* (PL) features extracted from the code using BERT-based models pre-trained on code corpora with NL features learned by RoBERTa (a pre-trained natural language model) from textual information. In the second approach `doc2vec` is first trained on the codes (of the dataset) to extract a distributed semantic representation of a code from the PL token sequence and then an NL-based representation of the textual information of a source code is generated by a `doc2vec` model trained on the comments. In the experiments performed using both approaches, in a within-project setting, on the Apache Calcite (Begoli et al., 2018) dataset, we aim to investigate if the code comments bring meaningful information besides the code itself. Another goal of the study is the comparison of the performance of an SDP predictor using the proposed features with approaches based on traditional software metrics-based features. To the best of our knowledge, a study similar to ours has not been conducted in the SDP literature. The research questions targeted in our work are as follows:

RQ1. Does the fusion of the natural language features (from code comments) and the programming language features (from code), learned using NL models (doc2vec, BERT-based), improve the performance in the SDP task?

RQ2. To what extent does the use of the semantic features automatically extracted from the code and/or code comments enhance the predictive performance of defect classifiers compared to the traditional software metrics-based features?

The paper is structured as follows. In Section 2 we review recent approaches, based on semantic features, from the SDP literature. Section 3 contains the description of the methodology employed in the study. The results of the experiments conducted on the Calcite software are presented and discussed in Section 4 while Section 5 exposes the threats to the validity of the present study. Conclusions and directions for future research are presented in the last section.

2 RELATED WORK ON SEMANTIC FEATURES USED FOR SDP

Recent works from the SDP literature that use semantic features to improve the software defect prediction performance are presented.

(Abdu et al., 2022) conducted a comprehensive study (based on 90 scientific articles) of the SDP approaches using contextual and semantic features extracted from the source code. The main aspects presented in this survey are (1) the source code semantic representations (ASTs, graph-based representations,

token embeddings), (2) the deep learning techniques applied (LSTM, DBN, CNN, BERT-based) and (3) the datasets (labeled and unlabeled) used in the semantic feature-based SDP models. The domain's critical problems and challenges are also described.

(Wang et al., 2016) proposed the first model based on semantic features using DBN (Deep Belief Network). The features learned from token vectors extracted from ASTs have been compared with 20 traditional software metrics-based features in the experiments conducted on ten open-source projects from the PROMISE dataset (Jureczko and Madeyski, 2010). In both within-project and cross-project SDP tasks, the semantic features proved to be more relevant than the other features. In the SDP model proposed by (Uddin et al., 2022) the contextual and semantic features of the source code are extracted from the tokens' semantic embeddings learned through a BERT model (Mohammed and Ali, 2021) and using the BiLSTM method (Graves et al., 2005) that exploits the contextual information. A data augmentation strategy has been employed to generate a large set of Java source codes, used further in fine-tuning the BERT model. The experiments conducted on the PROMISE dataset demonstrated that the proposed model outperformed both traditional SDP models and other DL models. (Miholca et al., 2022) proved that the features which capture the semantics of the source code are more informative than the software metrics-based features (Hosseini et al., 2019; Malhotra, 2015) in discriminating between defective and non-defective software entities. Two natural language–based models, `doc2vec` and LSI, have been used to extract contextual and semantic features of the class entities of 16 versions of the Calcite system. The DL-FASTAI model applied to these semantic features achieved the best results compared to other machine learning models (XGBoost, SVM, ANN) and other software metrics-based features.

Two types of features, internal and external, joined together with different weights, are used by (Zhou et al., 2022) to implement an SDP model. The internal semantic features are learned from each source file, using a CNN applied to the AST of the source code. The external features represent the structural information between all class files and are extracted from the software network (Class Dependency Network) using a GCN (Graph Convolutional Network). A large set of experiments was performed on the seven projects of the PROMISE dataset, in three settings: within-version, cross-version and cross-project. All the performance results expressed by AUC, F1 and accuracy metrics were satisfactory, but the best results were obtained for within-version prediction, using Synthetic Minority Oversampling Technique (SMOTE) and Random Forest (RF) as the classifier.

(Huo et al., 2018) proposed CAP-CNN (Convolutional Neural Network for Comments Augmented Programs) aiming to obtain a more informative semantic representation of code structure and functionality using also the comments. Semantic code features extracted by a CNN are concatenated with features extracted from code comments using another CNN and further used in defect prediction. To deal with cases of missing comments, the embeddings of tokens in the comments are learned during the training process. The experiments on the PROMISE dataset indicated that this model outperforms state-of-the-art methods and proved that the use of comments is beneficial in improving defect prediction. The semantic representation of a source code introduced by (Yao et al., 2023), is based on two types of features extracted from the code processed as a programming language entity and as a natural language text, respectively. The code syntax features were learned from ASTs using a Tree-based CNN method, and the code text features were extracted using an attention mechanism from the code token sequence after removing comments and function descriptions. The proposed program semantic feature mining method evaluated on the PROMISE dataset proved to outperform other DL-based SDP models in terms of F-measure.

Recently, the PM2-CNN (Pretrained Model-Based Multi-Channel Convolutional Neural Network) model was introduced by (Liu et al., 2023). Using UniXcoder (unified cross-modal pre-trained model for PL), the semantic PL features of the code and the NL features of the code description text are extracted simultaneously and then a multi-channel CNN processes both types of features to obtain rich semantic information, used further in defect prediction. The effectiveness of the model has been verified on Big-Vul, a large C/C++ dataset in the field of code vulnerabilities. Compared to four deep learning SDP methods, which do not use external textual information of the codes, PM2-CNN achieved better performance results in terms of precision, recall, and F1-score.

The present approach comes in the context of trying to improve the software defect prediction using additional textual information, such as comments. The novelty of our approach lies in the fusion of programming language features (from code) and natural language features (from code comments), learned using NL models (doc2vec or BERT-based) to obtain an enriched semantic representation of a source code.

3 METHODOLOGY

This section presents the methodology employed in the present study. We begin by describing the case study used in the paper, then the theoretical model for the SDP task is proposed. In Section 3.2 the feature-based representations used for the source code of application classes are introduced. In the next section the proposed SDP classifier and the experimental methodology are described.

The open-source framework for data management, Apache Calcite (Begoli et al., 2018), is the software system used in our study. Apache Calcite is an active open-source project, written in Java, with 330 contributors. The release cadence of the project on Maven Central is averaging 4 releases per year (between the years 2014 and 2023). Even if the project is continuously evolving, in this paper we are considering the first 16 Calcite releases, as presented by (Herbold et al., 2022). During releases 1.0.0-1.15.0, a total number of 1644 application classes were developed and maintained. For all application classes from each release from 1.0.0 to 1.15.0 of the Calcite software the ground truth label ("+" or "-") is available, as introduced by (Herbold et al., 2022). The label indicates the defective ("+") or non-defective ("-") property of an application class in that particular release. A severe data imbalance is observed in the dataset, the defective rate decreasing from 0.166 in the first release (178 defects and 897 non-defects) to 0.033 (45 defects and 1307 non-defects) in release 1.15.0. This dataset was also used by (Miholca et al., 2022), (Ciubotariu et al., 2023), (Briciu et al., 2023).

3.1 Theoretical Model

The SDP task is modeled as a binary classification problem, with two output classes: the positive class (denoted by "+") containing the software defects and the negative class (denoted by "-") containing non-defects.

The dataset, an object-oriented software system, is represented by $\mathcal{S} = \{(ac_1, l_1), \ldots, (ac_n, l_n)\}$, consisting of all the software application classes and the corresponding ground truth labels ("+" or "-"). In the present approach we consider two types of features: PL features learned from the code of the application class, $\mathcal{F}_{\mathcal{PL}} = \{fpl_1, \ldots, fpl_m\}$ and NL features extracted from the comments attached to the application class, $\mathcal{F}_{\mathcal{NL}} = \{fnl_1, \ldots, fnl_p\}$. Two sets of features $\mathcal{F}_{\mathcal{PL}}$ and $\mathcal{F}_{\mathcal{PL}} \cup \mathcal{F}_{\mathcal{NL}}$ are extracted and used separately to represent the application classes as high-dimensional vectors.

The SDP classifier is built from a labeled training dataset, using a feature-based representation of the application classes and predicts a label ("+" or "-") for a tested class.

3.2 Data Representations

For extracting semantic features characterizing the application classes and thus answering research question RQ1 we are considering embeddings obtained separately for both: (1) the *code* itself and (2) the *code comments*, concatenating the resulting representations to be used as input (referred in the following as **code+comments**). We hypothesise that the comments (represented by NL-based features), besides the code itself (represented by PL-based features), bring additional semantic information which would help increase the performance in the SDP task.

To achieve this, the code is separated from the comments using the pyparsing[1] module, and different processing pipelines are defined for each, depending on the embedding model. We note that the comments considered in our work are both *implementation comments* (delimited in the code by /* ... */ and //) and *documentation comments* which are Java-only, and are delimited by /**...*/.

Two types of embedding models are considered for the representation of the application classes, for both the code and the comments attached. First, we consider the semantic representations provided by a `doc2vec` model trained on the considered dataset (either on the PL or the NL content). The relevance of these features in an SDP task is compared to that of representations obtained using pre-trained BERT-based models, specifically CodeBERT- and RoBERTa-based ones. The decision to select these models was based on prior research demonstrating their suitability for the SDP task (Miholca et al., 2022), (Briciu et al., 2023), with the current work aiming to expand upon previous investigations by additionally examining the role of comments in differentiating between defect and non-defect classes.

In terms of data pre-processing, in order to obtain the `doc2vec` representations, a code tokenizer[2] is used to obtain the code tokens, while for the comments written in natural language, we used the nltk library[3] for tokenization. As far as code is concerned, the tokenizer identified whitespace and end-of-file as separate tokens, but these were eliminated as we considered that given the strategy employed by `doc2vec` in learning, including them may hinder the model's ability to learn meaningful document representations.

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

[2] https://github.com/Ikuyadeu/CodeTokenizer

[3] https://www.nltk.org/

For comments, no additional pre-processing was employed.

As for the BERT-based representations, the default model tokenizer, based on a Byte-Pair-Encoding algorithm, was used. The maximum length of a sequence is set to 512, which represents the upper limit of the model, to include as much information as possible from the input files, and truncation and padding are applied to the input code or comment instances to account for their varying lengths.

3.2.1 `doc2vec`-Based Representations

The *Paragraph Vector* model, or `doc2vec`, is an unsupervised learning algorithm that learns continuous, distributed fixed-length representations, called embeddings, for texts of varying lengths, such as sentences, paragraphs, or documents (Le and Mikolov, 2014). These representations are obtained by extending the neural network-based framework defined for the `word2vec` model to include a paragraph vector. In the Distributed Memory Model of Paragraph Vector (PV-DM), the paragraph vector and the word vectors are combined by either concatenation or averaging and asked to contribute to the task of predicting the next word in a text window. It is important to note that the PV-DM model takes into account word order and preserves the semantic relationships between words in the vector space. In contrast, the Distributed Bag-of-Words version of Paragraph Vector (PV-DBOW) only uses the paragraph vector to predict randomly sampled words from a text window within that paragraph.

`Doc2vec` has been successfully used in the task of SDP (Miholca et al., 2022; Aladics et al., 2021), either on its own or in conjunction with other representations. In our experiments, we separately trained `doc2vec` models on both the PL and NL content, respectively, within the application class, using the gensim[4] library. For the training step, we varied the Paragraph Vector model used (PV-DM or PV-DBOW) and the size of the resulting vector representation (50, 100, 150 or 300). The training of the `doc2vec` model is done only on the training set, and the vector representations (embeddings) for the instances in the validation and test sets (as defined in Section 3.3) are obtained through inference.

3.2.2 BERT-Based Representations

In the case of BERT-based semantic representations, we consider three pre-trained models to extract representations for code: CodeBERT-base-MLM (Feng et al., 2020), CodeBERTa[5] and CodeBERT-Java (Zhou et al., 2023). The first of these, CodeBERT-base-MLM, is initialized from a RoBERTa-base model and further trained on the CodeSearchNet code corpus with a MLM (Masked Language Modeling) objective. CodeBERTa, specifically CodeBERTa-small-v1, is another RoBERTa-like model, but with the architecture of DistilBERT. This model is also trained on the full CodeSearchNet corpus. Lastly, CodeBERT-Java builds upon the CodeBERT-base-MLM model, being additionally trained on Java code from the CodeParrot Github dataset on a MLM task. The choice of the models was driven by a comparison between the relevance of features obtained using CodeBERT-base-MLM, a model that has been used in a similar setting, and a model based on a different transformer architecture (CodeBERTa-small-v1) and one which was additionally trained on Java code (CodeBERT-Java).

To obtain semantic representations of natural language comments, we employed RoBERTa-base (Liu et al., 2019), a model that improves upon the original BERT model by using a dynamic masked language modeling training strategy and larger volumes of text.

The procedure for extracting the semantic representations (embeddings) using these BERT-based models does not depend on the type of input (i.e. code or natural language text in the form of specifications and comments). After completing the pre-processing steps outlined in Section 3.2, the tokenized programming language code or natural language text serves as input for a pre-trained model. A 768-dimensional representation corresponding to the entire input document is derived by applying a mean pooling technique to the 512 embeddings (for the tokens in the input) obtained from the last hidden state of the model. By opting for feature extraction instead of fine-tuning with BERT models, we can directly assess and compare the discriminatory power of the two types of representations, as they will both serve as input for the same SDP classifier.

3.3 SDP Classifier

The machine learning model we decided to use for the SDP task is the *eXtreme gradient boosting* (XGB) classifier, as it provides good performance in a wide range of tasks (Chen and Guestrin, 2016). The XGBoost classifier will be trained on the semantic representations of the application classes from the training data. In order to evaluate the practical capabilities of the SDP classifier, we use the application

[4]https://radimrehurek.com/gensim/models/doc2vec.html

[5]https://huggingface.co/huggingface/CodeBERTa-small-v1

classes (code or code+comments) from versions 0..14 for training the SDP classifier, followed by testing on version 15. In addition, the data from versions 0..14 is divided randomly in 80% for training the model and the remaining 20% for validation, a subset which is used for hyper-parameter optimization.

For assessing the performance of XGB on Calcite version 1.15.0, we are calculating several evaluation metrics used in forecasting (Mazzarella et al., 2017) and SDP literature (Fawcett, 2006), (Boughorbel et al., 2017): *probability of detection* (*POD*), *specificity* (*Spec* or true negative rate), *false alarm ratio* (*FAR*), *critical success index* (*CSI*), *Area under the ROC curve* (*AUC*), *Matthews Correlation Coefficient* (*MCC*) and *F-score for the positive class* ($F1$). The classification results expressed by the following values: TP - number of true positives (the defects correctly classified), FP - number of false positives (the non-defects incorrectly predicted as defects), TN - number of true negatives (the non-defects classified correctly), FN - number of false negatives (the defects incorrectly classified as non-defects) are computed on the testing set. Then, the metrics values are calculated using the formulas: $POD = \frac{TP}{TP+FN}$, $Spec = \frac{TN}{TN+FP}$, $FAR = \frac{FP}{TP+FP}$, $CSI = \frac{TP}{TP+FN+FP}$, $AUC = \frac{POD+Spec}{2}$, $MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP+FP)\cdot(TP+FN)\cdot(TN+FP)\cdot(TN+FN)}}$, and $F1 = \frac{2 \cdot TP}{2 \cdot TP+FP+FN}$. The *MCC* metric ranges in [-1, 1] while the other measures take values in [0, 1]. Lower values for *FAR* indicate better predictors, while for the other metrics higher values are better. We note that *AUC* is considered in the SDP literature one of the best metrics for evaluating the performance of the defect classifiers (Fawcett, 2006).

4 RESULTS AND DISCUSSION

This section presents the results of our experiments aimed to answer the research questions RQ1 and RQ2.

4.1 Results

As previously discussed, our experiments consist of training the XGB classifier (described in Section 3.3) on the Calcite versions from 1.0.0 to 1.14.0 and afterwards evaluating its performance on release 1.15.0. Through the proposed evaluation we aim to simulate a real life scenario where an existing project is further developed and all the existing historical data in the project is used to improve the next release of the software through the use of SDP tools.

Multiple experiments were performed using both `doc2vec` and BERT-based semantic representations, varying the representation size and PV model (`doc2vec`) and the side from which the input is truncated when it is longer than the maximum length of the sequence (BERT-based representations). The best results obtained throughout these experiments, using the representations detailed in Section 3.2 and applied on the two possible inputs (**code** and **code+comments**) are illustrated in Tables 1 and 2. For each experiment conducted, the obtained classification results are presented, together with the values for the performance metrics used for evaluation. The best value is highlighted for each specific model and each performance metric.

To answer the research question **RQ1** we introduce an evaluation measure, denoted by *score*. The *score* is computed for each of the two representations r employed in the experiments: the representation learned using `doc2vec`-based models (r = `doc2vec`) and the representation learned using BERT-based models (r = BERT). For such a representation r, considering all the performed experiments (depicted in Table 1 for r = `doc2vec` and in Table 2 for r = BERT), we define $score(r) = \frac{n(r)}{21}$. In the previous formula, 21 is the total number of cases (i.e., seven performance metrics for each of the three performed experiments) and $n(r)$ is the number of performance metrics whose values are better for the **code+comments** input than for the **code** input. Thus, $score(r)$ represents the percentage of the cases that show an improvement of the SDP performance when fusing the semantic features extracted from code with those extracted from code comments, compared to the semantic features extracted solely from the code.

Table 3 presents the values of *score* for both the `doc2vec` and BERT-based representations. The positive values obtained for the *score* highlight that the addition of the comments embedding to the code embedding is beneficial in SDP.

For a wider perspective, we can consider all the performed experiments, which involved, for BERT, experimenting with a left-side truncation besides the default right-side one, and for `doc2vec`, examining various combinations of models and representation sizes (as detailed in Section 3.2.1). Table 4 shows the values obtained for each evaluation metric when averaged over all performed experiments together with the 95% confidence interval of the mean values. As can be observed, the inclusion of information found in NL comments alongside the code encoding yields benefits, enhancing the overall performance of the models. Among the 14 comparisons made, the configuration that incorporates both code and comments as input wins in 11 instances, constituting 79% of the cases.

Table 1: Experimental results obtained using doc2vec-based models for generating code and comments embeddings. The classification results calculated on Calcite 1.15.0 and the performance metrics values are presented.

Experiment	Model	Input	Embedding length	TP	FP	TN	FN	*POD* (↑)	*Spec* (↑)	*FAR* (↓)	*CSI* (↑)	*AUC* (↑)	*MCC* (↑)	*F*1 (↑)
1	**PV-DBOW**	**code**	50	35	24	1283	10	**0.778**	**0.982**	**0.407**	**0.507**	**0.880**	**0.667**	**0.673**
		code+comments	50+50	33	30	1277	12	0.733	0.977	0.476	0.440	0.855	0.605	0.611
2	**PV-DM**	**code**	150	34	41	1266	11	**0.756**	0.969	0.547	0.395	0.862	0.568	0.567
		code+comments	150+100	34	31	1276	11	**0.756**	**0.976**	**0.477**	**0.447**	**0.866**	**0.614**	**0.618**
3	**PV-DM**	**code**	300	32	35	1272	13	0.711	0.973	0.522	0.400	0.842	0.566	0.571
		code+comments	300+100	34	30	1277	11	**0.756**	**0.977**	**0.469**	**0.453**	**0.866**	**0.619**	**0.624**

Table 2: Experimental results obtained using BERT-based models for learning code and comments embeddings. The classification results calculated on Calcite 1.15.0 and the performance metrics values are presented.

Experiment	Model	Input	TP	FP	TN	FN	*POD* (↑)	*Spec* (↑)	*FAR* (↓)	*CSI* (↑)	*AUC* (↑)	*MCC* (↑)	*F*1 (↑)
1	**CodeBERT-MLM**	**code**	35	26	1281	10	**0.778**	**0.980**	**0.426**	**0.493**	**0.879**	**0.655**	**0.660**
	CodeBERT-MLM + RoBERTa	**code+comments**	35	33	1274	10	**0.778**	0.975	0.485	0.449	0.876	0.618	0.619
2	**CodeBERT-Java**	**code**	34	27	1280	11	0.756	**0.979**	0.443	0.472	0.867	0.635	0.642
	CodeBERT-Java + RoBERTa	**code+comments**	37	28	1279	8	**0.822**	**0.979**	**0.431**	**0.507**	**0.900**	**0.671**	**0.673**
3	**CodeBERTa-small**	**code**	36	21	1286	9	**0.800**	**0.984**	**0.368**	**0.545**	**0.892**	**0.700**	**0.706**
	CodeBERTa-small + RoBERTa	**code+comments**	36	21	1286	9	**0.800**	**0.984**	**0.368**	**0.545**	**0.892**	**0.700**	**0.706**

Table 3: Values obtained for the *score* measure and the doc2vec and BERT-based representations employed.

Representation	*score*
doc2vec-based	**0.62**
BERT-based	**0.29**

Thus, overall, the values of the performance metrics are better for PL+NL feature representation than using only PL features.

In terms of a comparison between the two types of embeddings, the BERT-based representations yield consistently better results than the doc2vec embeddings, despite the fact that in the tokenization process, these models reduce all input sequences to a length of 512. This result underlines the importance of the information encoded in the pre-training step of the BERT-based models using large source code datasets.

For doc2vec, the best results are obtained when considering only the code as input, with a representation of size 50 obtained using the PV-DBOW architecture. Since the PV-DBOW model does not account for word order, using only the paragraph vector to predict words in a text window, it captures document-level semantics more than it does local context, which proves useful in separating defects and non-defects. However, in our experiments, we observed that as the size of the representation obtained with the PV-DBOW model increases, its ability to identify defects decreases (for size of 100 or 150, for instance, a *POD* value of 0.622 is obtained, while for size 300, the *POD* value decreases further, to 0.600). In contrast, the *Specificity* steadily increases with representation size. An explanation for this could refer to the substantial imbalance in the number of non-defective instances relative to defective ones: the increased size of representation allows for more comprehensive coverage of diverse document-level patterns indicative of a non-defect. For PV-DM, which also encodes word-order information, a size of 150 provides the best results in terms of the *POD* metric (0.756), but similarly to the PV-DBOW model, increasing the size of the representation aids in the identification of non-defects, as shown by the increased value for *Specificity* for a representation size of 300.

Concatenating the representations obtained for code and those obtained for the comments included in the files alongside it leads to better results if the code representation does not manage to capture sufficient relevant information to discriminate between defects and non-defects on its own, as highlighted by the results in Tables 1 and 2. For comments, the results indicate that a representation size of 100 is best, managing to improve upon different source code representations in multiple testing configurations, especially as far as the identification of non-defects is concerned.

The improvement brought by adding information encoded in the comment embeddings is less evident in the case of BERT-based representations. This could be attributed to the operation of truncation to 512 tokens of the natural language input sequence applied by RoBERTa, similar to the truncation of the programming language code. In terms of a comparison across these models, the CodeBERT-Java model for code embeddings in conjunction with natural language text embeddings obtained using RoBERTa-base provides the best result for the *AUC* metric (0.9).

When employing doc2vec, a first challenge lies in determining the optimal representation size that effectively captures relevant patterns from both code and natural language comments, respectively. Additionally, identifying the combination that yields the most favorable results adds further complexity, as the number of experiments to be performed for an exhaustive search is quite large. Secondly, the inference step in

Table 4: Average obtained over all performed experiments for the evaluation metrics for the two types of input and the two embedding models. 95% confidence intervals are used for the results.

Representation	Input	*POD* (↑)	*Spec* (↑)	*FAR* (↓)	*CSI* (↑)	*AUC* (↑)	*MCC* (↑)	*F*1 (↑)
doc2vec	code	0.664±0.050	**0.980±0.005**	0.461±0.045	0.420±0.029	0.822±0.024	0.580±0.033	0.589±0.033
	code+comments	**0.706±0.022**	0.979±0.002	**0.459±0.027**	**0.440±0.019**	**0.843±0.011**	**0.602±0.018**	**0.610±0.018**
BERT-based	code	0.715±0.063	**0.983±0.002**	**0.407±0.023**	0.479±0.035	0.849±0.031	0.637±0.034	0.646±0.032
	code+comments	**0.733±0.033**	0.982±0.002	0.415±0.029	**0.482±0.026**	**0.858±0.016**	**0.641±0.025**	**0.649±0.024**

the `doc2vec` model is not inherently deterministic, yielding similar but not identical representations for the same input. This may pose a problem in the current testing configuration, as the content of some files across the versions does not change, but their encoding does, even if marginally.

A limitation of the proposed BERT-based representations refers to the truncation performed in the tokenization step of these models, which limits the number of tokens to 512. Therefore, modifications in the code may not be taken into consideration across versions if the modification is in the part of the code that is truncated. Possible solutions to this issue refer to either the use of other transformer-based models that are better able to handle long input sequences (Beltagy et al., 2020) or the examination of other types of features in conjunction with the current `doc2vec` and BERT-based representations, strategy which has proved winning in previous works that address the SDP task (Miholca et al., 2022).

4.2 Discussion

While we acknowledge that limitations in the proposed approach may be one reason for misclassified instances, this section aims to analyze these instances to illustrate other potential causes for misclassification. We further present some concrete examples from the analysed project, that are not sufficient in isolation to conclude that the performance of our proposed method would be higher, but are an indication that any SDP approach should also consider the reality of the considered software project and details about the actual source code in order to better understand the larger context of the changes made in the project. The dataset (Herbold et al., 2022) used for the case study provides labels for Apache Calcite up to version 1.1.15 but the project is in a continuous development, so we analysed the changes made to the source code after this release.

For better understanding the misclassifications provided by our XGB classifier we analyzed both the source code and the commit messages from Git for the application classes that were incorrectly classified. We mention that the analysis was performed on the best performing model, namely XGB trained on embeddings for the applications classes automatically learned using CodeBERT-Java from the code concatenated with the comments embedding generated through RoBERTa.

We investigated the application classes from Calcite 1.15.0 that were classified as defective but were labeled non-defective in the original dataset (Herbold et al., 2022) and we noticed that multiple changes were performed in the source code after the release of the version 1.15.0. Commit messages were not part of the data used for training the models, but we analyzed them in other to get a better understanding of the changes performed on the misclassified classes. While some of the commits are clearly not related to bug fixes (i.e., the commit message starts with expressions like: "Refactor:","Code style:","Update formatting:" some commit messages clearly indicate that the commit is related to a code change that is a fix for a defect. For example, the commit with the ID `ccbacf6` and the commit message "`RelDataType CACHE in RelDataTypeFactoryImpl can't be garbage collected`" seems to fix a memory leak issue. Our classifier predicted the class `RelDataTypeFactoryImpl` as being defective, even if the class was labeled as being non-defective. Thus, it is very likely that the classification is correct, as the code that is changed in this particular commit predates version 1.15.0 and this indicates that the class actually contained a defect but it was undetected at the time the labels were generated.

Analysing the commit messages for the class `org.apache.calcite.rex.RexSimplify` that was predicted as non-defective by our approach but labeled as defective in the dataset, we found contradicting information in the commit messages. For instance, we found the following three commit messages: (1) Message in commit `141781b`: "`Avoid simplification of x AND NOT(x) to false for nullable x`"; (2) Message in commit `c8e91ea`: "`RexSimplify: AND(x, y, NOT(y)) ⇒ AND(x, null, IS NULL(y))`"; and (3) Message in commit `90f49be`:"`RexSimplify should optimize '(NOT x) IS NULL' to 'x IS NULL' (pengzhiwei). Previously it optimized '(NOT x) IS NULL' to 'x IS NOT NULL', which is wrong.`". All three previous commits change only the method `simplifyIs2(SqlKind kind, RexNode a)` in the class `RexSimplify` and the messages suggest

that there is not a clear understanding on what the specification and requirements for this method should be. Thus, it is unclear if the application `org.apache.calcite.rex.RexSimplify` is indeed a misclassification (i.e., if the defective label assigned to this class in Calcite 1.15.0 is correct).

4.3 Comparison to Related Work

To answer the research question **RQ2**, we are further providing a comparison with approaches from the literature which performed within-project SDP on the Calcite software using software metrics or/and semantic features for representing the application classes.

Three related approaches from the literature will be further considered. The first related approach is the one introduced by (Ciubotariu et al., 2023). The authors adapted the *support vector* classifier model to an one-class classification model (OCSVM) to perform outlier detection. Experiments were conducted on Apache Calcite (Begoli et al., 2018), by training the OCSVM model on both the "+" (defective) and "-" (non-defective) classes. The performance of the OCSVM models was compared with the performance of the binary SVC model. As in our experiments, the models were trained on all Calcite releases from 1.0.0 to 1.14.0 and tested on the 1.15.0-th release, thus allowing a direct comparison between the results. We note that we computed the values for the *MCC* metric from the confusion matrices provided in (Ciubotariu et al., 2023). The second related approach is the one introduced by (Miholca et al., 2022). The relevance of both semantic features extracted from the code of the application classes using `doc2vec` and LSI language models and software metrics-based features (Herbold et al., 2022) is comparatively analyzed. The experimental evaluation is conducted on the available releases of the Calcite software. The testing methodology slightly differs from ours, as a cross-validation (CV) is performed on each Calcite release. For a more precise comparison, we will use the result provided by (Miholca et al., 2022) after applying a 5-fold CV on Calcite version 1.15.0. Another related work considered in our comparison was introduced by (Briciu et al., 2023). Experiments were conducted on the Calcite software using the same training and testing methodology as in the current paper (the first 15 releases were used for training, while testing was performed on release 1.15.0). The application classes from the Calcite software represented as embeddings learned from the source code using the CodeBERT-base-MLM model were used for training an artificial neural network (ANN) classifier.

Table 5 comparatively presents the results obtained in our approach and the results achieved by the previously described approaches. The table depicts the SDP classifier (ML model) employed, the features employed for representing the application classes and the evaluation measures used. Most of the performance metrics employed in our paper were employed in the related approaches as well. We will mark in the table with "–" the cells corresponding to performance metrics for which the values were not reported in the related work. We note that we are considering for comparison our best performance (as shown in Table 2, the fourth row) provided by the XGB classifier with embeddings for the application classes automatically learned using CodeBERT-Java from the code concatenated with the comments embedding generated through RoBERTa.

A comparative analysis of the results depicted in Table 5 reveal the following. As a first conclusion, our proposed semantic representation of the application classes, obtained from the embedding learned using CodeBERT-Java from the code combined with comments embedding generated with RoBERTa, leads to a higher performance than the semantic representation using `doc2vec`+LSI proposed by (Miholca et al., 2022) in terms of all evaluation metrics. The XGBoost classifier has a sligthly higher specificity than our model (0.993 vs. 0.979), but it has a very low *POD* (0.166 vs. 0.822). An improvement of more than 40% is observed in the recall of our classifier (*POD*) highlighting a better defect detection rate. In addition, the *MCC* metric which is considered in the literature a good evaluation metric for imbalanced datasets is also improved with more than 37%. With respect to the BERT-based representation used in (Briciu et al., 2023), which encodes only code information, the fusion of comments and code embeddings provides superior results in all metrics with the exception of *POD*. This can be explained by the ANN classifier in the study employing class weights and thus assigning higher importance to the correct classification of defective instances, leading to a high *POD*, but low *MCC*, *CSI*, and $F1$ values. Our proposed representation, in conjunction with an XGB classifier, leads to significantly higher scores in terms of the latter metrics, while still obtaining comparable results with respect to *POD*. The statistical significance of the improvement in terms of the *POD*, *Spec*, *FAR*, *AUC*, *MCC* and $F1$ metrics was confirmed using a one-tailed paired Wilcoxon signed-rank test (Wilcoxon Signed-Rank Test, 2023).

The second conclusion of the conducted comparison to related work is the answer to the research question **RQ2**. We observe that the fusion of PL features

Table 5: Comparison to related work.

Model	Features / Model	*POD* (↑)	*Spec* (↑)	*FAR* (↓)	*CSI* (↑)	*AUC* (↑)	*MCC*(↑)	*F*1 (↑)
$OCSVM_+$ (Ciubotariu et al., 2023)	Software metrics	0.644	0.523	0.956	0.043	0.584	0.060	0.083
$OCSVM_-$ (Ciubotariu et al., 2023)	Software metrics	0.711	0.533	0.950	0.049	0.622	0.088	0.093
SVC (Ciubotariu et al., 2023)	Software metrics	0.711	0.963	0.600	0.344	0.837	0.514	0.512
DL-FASTAI (Miholca et al., 2022)	Software metrics	0.245	0.979	0.613	–	0.612	0.272	0.276
XGBoost (Miholca et al., 2022)	Software metrics	0.166	0.993	0.443	–	0.580	0.279	0.242
DL-FASTAI (Miholca et al., 2022)	Semantic features / `doc2vec`+LSI	0.390	0.961	0.726	–	0.675	0.291	0.308
ANN (Briciu et al., 2023)	Semantic features / CodeBERT-base-MLM	0.899	0.884	0.792	0.203	0.886	0.397	0.338
Proposed XGB classifier	Semantic features / CodeBERT-Java + RoBERTa	**0.822**	**0.979**	**0.431**	**0.507**	**0.900**	**0.671**	**0.673**

extracted from the code and the NL features from comments improves the performance of software defect predictors compared to the feature-based representation using software metrics. The SDP classifier considering embeddings learned using CodeBERT-Java from the code and RoBERTa from the comments (the last line from the table) outperforms the other classifiers (lines 1-5 from the table) that use software metrics-based representation of the application classes. The binary SVC classifier proposed by (Ciubotariu et al., 2023) has the closest *AUC* value (0.837 vs. 0.900), but has a worse *FAR* (0.600 vs. 0.431) suggesting a high rate of false positive results.

5 THREATS TO VALIDITY

The threats to the validity of our study and biases that may impact it are further discussed (Runeson and Höst, 2009). Regarding *construct validity*, we have employed relevant performance metrics for evaluating the performance of the defect classifier. Best practices in building, testing and evaluating the learning models were used for reducing the threats to construct validity: model validation during training, a testing scenario which simulates the software evolution process, statistical analysis of the obtained results.

Threats to *internal validity* were minimized by exploring several models for each of the two distinct techniques to obtain the embeddings and examining different sets of parameters for fine-tuning the classification model. In what concerns the source code embeddings we observed that, in rare cases, the source code embeddings employed do not capture the change made to the source code which may have a small negative impact on the accuracy of the proposed approach. We analyzed the change history for classes whose labels were changed between releases (e.g., a class that was not defective in a release and labeled as defective in a new release or labeled defective in a release and labeled non-defective in the new release where the defect was fixed). In some cases (e.g. the class `org.apache.calcite.sql.type.SqlTypeUtil`), the cosine distance between the two embedddings (the one generated using the source code before and after the modification) is 0 indicating that no semantic difference between the two versions was identified by the models used for generating text embeddings (BERT, doc2vec). There can be many reasons for this, and further investigations are needed, but it's clear that other truncation and padding mechanisms need to be explored when pre-processing the inputs for the embedding model to guarantee that no important information is lost at this step.

In terms of *external validity* we have used as a case study in our work the open-source software Apache Calcite that is representative for a well-maintained, active, open-source project. An analysis of the change history for every application class in the Calcite releases 1.0.0 - 1.15.0 revealed some special cases that increased the difficulty of the software defect classification task, where an application class changed its label in the dataset (i.e., it was marked as *defect* in one version and as *non-defect* in another version) but without any change in the source code of the application class. Such cases may be due to mislabeling in the dataset, but most likely are caused by the nature of the evolution of any software system. For instance, some defects may not be discovered immediately and will be reported in future software releases. From a supervised classification viewpoint, such cases introduce noise during training, since an application class with exactly the same vectorial representation appears with two different labels in the dataset and may impact the model's performance. However, in the current work we did not remove these occurrences from the training dataset.

The dataset used in the experimental evaluation (Herbold et al., 2022) only covers releases between 1.0.0 and 1.15.0 but the Apache Calcite project continuously evolved and new releases are available. As a future work we plan to extend the dataset with labels

for the releases not covered by the public dataset and further validate the results obtained by the presented approach. To facilitate repeatability of the experiments presented in this paper, we provided in Section 3 sufficient details about the models employed and the experiments performed. Additionally, the text embedding vectors for both `doc2vec` and BERT-based representations are publicly available at (Briciu, A, 2024).

In what concerns the *reliability* and the *testing methodology* we created the experimental validation taking into account the actual evolution of the analyzed system. The model was created using the data available in 15 versions, from 1.0.0 to 1.14.0 and then was tested on the version 1.15.0. The performance of the model is assessed using multiple evaluation metrics and the statistical significance of the obtained results was analysed in order to increase confidence in the obtained results.

6 CONCLUSIONS AND FUTURE WORK

The aim of the present study was to analyze the relevance of the textual information of a code, such as comments, in increasing the SDP performance. To extract the semantic representation of a source code, two approaches were proposed. In the first approach, the pre-trained models CodeBERT and RoBERTa have been used for code embedding and textual embedding of the comments. In the second method a `doc2vec` model, trained on the codes, generates the semantic representation of a programming language code and then a `doc2vec` model trained on the comments generates the semantic representation of the textual information attached to the code. The conclusion drawn from the performed experiments, using both feature extraction methods, is that the addition of the NL-based features to the PL-based features has a positive impact in defect prediction. Also, these new types of semantic features proved to be more informative than the software metrics-based features.

However, different development methodologies and principles may impact the amount and quality of textual information in the source code. Even in methodologies that favor self-explanatory code instead of the use of comments (i.e., clean code principles), semantic textual information is still present in the code in form of variables, function names and specifications (usually embedded in the source code as comments). More work needs to be done in order to asses the impact of programming methodologies on the proposed approach, but semantic textual information needs to be present in any code base as long as the code is written to be understandable not only by the computer but also by other programmers. If not constantly maintained, code comments and specifications may become obsolete, misleading, or decoupled from the actual problem domain. Further experimentation is needed to assess the impact of the quality of comments and specifications on the performance of the proposed approach.

The evaluation experiments will be extended to the new versions of the Calcite system and for the cross-project setting. A future work direction is to identify an appropriate weighting scheme for combining PL-based and NL-based features to obtain a relevant semantic representation that better discriminates between defective and non-defective codes. The refinement of the binary classification in more types of defects, using unsupervised approaches and different types of features is another topic of further research.

REFERENCES

Abdu, A., Zhai, Z., Algabri, R., Abdo, H., Hamad, K., and Al-antari, M. (2022). Deep learning-based software defect prediction via semantic key features of source code—systematic survey. *Mathematics*, 10.

Aladics, T., Jász, J., and Ferenc, R. (2021). Bug prediction using source code embedding based on doc2vec. In *International Conference on Computational Science and Its Applications*, pages 382–397. Springer.

Begoli, E., Camacho-Rodríguez, J., Hyde, J., Mior, M. J., and Lemire, D. (2018). Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of SIGMOD '18*, page 221–230, New York, NY, USA. ACM.

Beltagy, I., Peters, M. E., and Cohan, A. (2020). Longformer: The long-document transformer.

Boughorbel, S., Jarray, F., and El-Anbari, M. (2017). Optimal classifier for imbalanced data using matthews correlation coefficient metric. *PloS one*, 12(6).

Briciu, A., Czibula, G., and Lupea, M. (2023). A study on the relevance of semantic features extracted using BERT-based language models for enhancing the performance of software defect classifiers. *Procedia Computer Science*, 225:1601–1610.

Briciu, A (2024). BERT-based and doc2vec representations. https://github.com/anamariabriciu/sdp-calcite-doc2vec-bert. Online; accessed 10 March 2024.

Chen, T. and Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In *Proceedings of KDD '16*, pages 785–794, New York, NY, USA. ACM.

Ciubotariu, G., Czibula, G., Czibula, I. G., and Chelaru, I.-G. (2023). Uncovering behavioural patterns of one- and binary-class SVM-based software defect predictors. In *Proceedings of ICSOFT 2023*, pages 249–257. SciTePress.

Clark, B. and Zubrow, D. (2001). How good is a software: a review on defect prediction techniques. In *Software Engineering Symposium*, pages 1–35, Carneige Mellon University.

Dam, H. K., Pham, T., Ng, S. W., Tran, T., Grundy, J., Ghose, A., Kim, T., and Kim, C. (2018). A deep tree-based model for software defect prediction. *CoRR*, abs/1802.00921:1–10.

Fawcett, T. (2006). An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. In *EMNLP 2020*, pages 1536–1547, Online.

Graves, A., Fernández, S., and Schmidhuber, J. (2005). Bidirectional LSTM networks for improved phoneme classification and recognition. In *Proceedings of Artificial Neural Networks: Formal Models and Their Applications - ICANN 2005*, pages 799–804.

Herbold, S., Trautsch, A., Trautsch, F., and Ledel, B. (2022). Problems with szz and features: An empirical study of the state of practice of defect prediction data collection. *Empirical Software Engineering*, 27(2).

Hosseini, S., Turhan, B., and Gunarathna, D. (2019). A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering*, 45(2):111–147.

Hryszko, J. and Madeyski, L. (2018). Cost effectiveness of software defect prediction in an industrial project. *Foundations of Computing and Decision Sciences*, 43(1):7 – 35.

hua Chang, R., Mu, X., and Zhang, L. (2011). Software defect prediction using non-negative matrix factorization. *Journal of Software*, 6(11):2114–2120.

Huo, X., Yang, Y., Li, M., and Zhan, D.-C. (2018). Learning semantic features for software defect prediction by code comments embedding. In *ICDM 2018*, pages 1049–1054.

Jin, C. (2021). Cross-project software defect prediction based on domain adaptation learning and optimization. *Expert Systems with Applications*, 171:114637.

Jureczko, M. and Madeyski, L. (2010). Towards identifying software project clusters with regard to defect prediction. In *The 6th International Conference on Predictive Models in Software Engineering*, page 1–10.

Le, Q. V. and Mikolov, T. (2014). Distributed representations of sentences and documents. *Computing Research Repository (CoRR)*, abs/1405.4:1–9.

Li, J., He, P., Zhu, J., and Lyu, M. R. (2017). Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security*, pages 318–328.

Liu, J., Ai, J., Lu, M., Wang, J., and Shi, H. (2023). Semantic feature learning for software defect prediction from source code and external knowledge. *Journal of Systems and Software*, 204:111753.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. (2019). RoBERTa: A Robustly Optimized BERT Pretraining Approach.

Malhotra, R. (2015). A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27:504 – 518.

Marcus, A., Poshyvanyk, D., and Ferenc, R. (2008). Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300.

Mazzarella, V., Maiello, I., Capozzi, V., Budillon, G., and Ferretti, R. (2017). Comparison between 3d-var and 4d-var data assimilation methods for the simulation of a heavy rainfall case in central italy. *Advances in Science and Research*, 14:271–278.

Miholca, D.-L., Tomescu, V.-I., and Czibula, G. (2022). An in-depth analysis of the software features' impact on the performance of deep learning-based software defect predictors. *IEEE Access*, 10:64801–64818.

Mohammed, A. H. and Ali, A. H. (2021). Survey of BERT (Bidirectional Encoder Representation Transformer) types. *Journal of Physics: Conference Series*, 1963(1):012173.

Runeson, P. and Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, 14(2):131–164.

Uddin, M., Li, B., Ali, Z., Kefalas, P., Khan, I., and Zada, I. (2022). Software defect prediction employing BiLSTM and BERT-based semantic feature. *Soft Computing*, 26:7877–7891.

Wang, S., Liu, T., and Tan, L. (2016). Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, pages 297–308. ACM.

Wilcoxon Signed-Rank Test (2023). Social science statistics. http://www.socscistatistics.com/tests/.

Yang, X., Lo, D., Xia, X., Zhang, Y., and Sun, J. (2015). Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26.

Yao, W., Shafiq, M., Lin, X., and Yu, X. (2023). A Software Defect Prediction Method Based on Program Semantic Feature Mining. *Electronics*, 12(7).

Zhao, L., Shang, Z., Zhao, L., Zhang, T., and Tang, Y. (2019). Software defect prediction via cost-sensitive siamese parallel fully-connected neural networks. *Neurocomputing*, 352:64–74.

Zheng, J. (2009). Predicting software reliability with neural network ensembles. *Expert Systems with Applications*, 36(2):2116–2122.

Zhou, C., He, P., Zeng, C., and Ma, J. (2022). Software defect prediction with semantic and structural information of codes based on graph neural networks. *Information and Software Technology*, 152:107057.

Zhou, S., Alon, U., Agarwal, S., and Neubig, G. (2023). CodeBERTScore: Evaluating code generation with pretrained models of code. In *Proceedings of the 2023 Conf. on Empirical Methods in Natural Language Processing*, pages 13921–13937, Singapore. ACM.

Zhu, K., Zhang, N., Ying, S., and Wang, X. (2020). Within-project and cross-project software defect prediction based on improved transfer naive bayes algorithm. *Computers, Materials & Continua*, 63(2):891–910.