Received May 12, 2022, accepted May 29, 2022, date of publication June 10, 2022, date of current version June 23, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3181995

An in-Depth Analysis of the Software Features' Impact on the Performance of Deep Learning-Based Software Defect Predictors

DIANA-LUCIA MIHOLCA, VLAD-IOAN TOMESCU, AND GABRIELA CZIBULA
Department of Computer Science, Babes-Bolyai University, 400347 Cluj-Napoca, Romania

Corresponding author: Diana-Lucia Miholca (diana.miholca@ubbcluj.ro)

This work was supported by the Ministry of Research, Innovation and Digitization, CNCS/CCCDI–UEFISCDI, within PNCDI III, under Project PN-III-P4-ID-PCE-2020-0800.

ABSTRACT *Software Defects Prediction* represents an essential activity during software development that contributes to continuously improving *software quality* and software maintenance and evolution by detecting defect-prone modules in new versions of a software system. In this paper, we are conducting an in-depth analysis on the software features' impact on the performance of deep learning-based software defect predictors. We further extend a large-scale feature set proposed in the literature for detecting defect-proneness, by adding conceptual software features that capture the semantics of the source code, including comments. The conceptual features are automatically engineered using Doc2Vec, an artificial neural network based prediction model. A broad evaluation performed on the Calcite software system highlights a statistically significant improvement obtained by applying deep learning-based classifiers for detecting software defects when using conceptual features extracted from the source code for characterizing the software entities.

INDEX TERMS Deep learning, Doc2vec, latent semantic indexing, software defect prediction.

I. INTRODUCTION

Software Defects Prediction (SDP) consists in identifying defective software components, being considered an essential activity during software development. It represents the activity of identifying defective software modules in new versions of a software system [1]. SDP is considered of great importance in software engineering, as it contributes to continuously improving the *software quality*. Developing high quality software systems is expensive and, in this context, SDP is used for increasing the cost effectiveness of quality assurance and testing [2]. By detecting fault-prone modules in new versions of a software system, SDP helps to allocate the effort so as to test more thoroughly those modules [1].

SDP assists measuring project evolution, supports process management [3], predicts software reliability [4], guides testing and code review [1]. All these activities allow to significantly reduce the costs involved in developing and maintaining software products [5]. Moreover, particularly in the case of safety-critical systems, SDP helps in detecting software anomalies with possible negative effects on human lives.

As the software systems complexity increases, the number of software defects generated during the software development will also significantly increase. This growing complexity of software projects requires an increasing attention to their analysis and testing. Numerous researches from the SDP literature are based on mining historical and code information during the software development process and then building a prediction model (statistical, machine learning-based or other) to predict software defects [6].

Despite its importance and extensive applicability, SDP remains a difficult problem, especially in large-scale complex systems, and a very active research area [7]. The conditions for a software module to have defects are hard to identify and, therefore, the defect prediction problem is computationally difficult. From a supervised learning viewpoint, predicting defects is a difficult task as the training data used for building the defect predictors is highly imbalanced. The faulty modules in a software system are greatly outnumbered

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

by the error-free modules. Therefore, conventional learning algorithms are often biased towards the non-defective class. Another important issue in SDP is related to the features used for characterizing software entities (an entity may be a component, class, module – depending on the targeted level of granularity). As, generally, in *machine learning* (ML), the classical approach is to use manually engineered features, traditional software metrics are usually used in SDP as features characterizing the given software entities. Literature reviews in SDP revealed that about 87% [8] of the case studies used procedural or object-oriented software metrics.

The two prevalent research directions in the SDP literature are: proposing software features relevant to the discrimination between defective and non-defective software entities and building or recommending high-performing defects prediction models.

When it comes to large amounts of data, deep learning models are some of the best at making accurate predictions, regardless of the origin of that data. As long as there is correlation between the input information and the output, the models will discover it. In order to use deep learning, the input software features are written in tabular form, a data form that has been extensively researched and for which many models are available [9].

In the present work, we follow both above-mentioned directions. Our study originated from three research questions:

RQ1 Could the performance of predicting software defects be enhanced by enlarging the software features proposed for SDP with conceptual features extracted from the source code? Which is the most appropriate feature set to distinguish between defective and non-defective software entities and to what extent is the performance improvement significant from a statistical perspective?

RQ2 Could the relevance of the conceptual-based software features be empirically sustained by both unsupervised and supervised analyses conducted on a large scale software system?

RQ3 Does deep learning-based defect prediction bring a statistically significant improvement when compared to traditional supervised classifiers?

With these research questions in mind, we have performed an in-depth analysis of the software features' impact on the performance of software defect predictors. We have extended the large collection of SDP features proposed by Herbond *et al.* [10] with Doc2Vec and LSI-based conceptual software features that capture the semantics of the source code (including comments). An extensive study conducted on different versions of the Calcite data set highlight, through both unsupervised and supervised learning-based analyses, that the conceptual features bring a statistically significant improvement on the performance of SDP. As a second line of research, we have extensively examined the effect of the feature set identified as being the most relevant on the performance of various defect predictors. To the best of our knowledge, a study similar to ours has not been proposed in the literature, so far.

The remainder of the paper is organized as follows. Section II states and formalises the SDP problem, highlighting its importance and practical relevance. An extensive review on existing machine learning-based approaches for predicting software defects as well as the data sets and features used in the SDP literature are presented in Section III. Section IV presents the experimental data and methodology used in our work. Section V details the first stage of our research, which consists in a machine learning-based analysis of the software feature sets' relevance. The performances obtained when applying various defect prediction models are comparatively analysed in Section VI. The threats to validity are discussed in Section VII, while Section VIII highlights the conclusions of our paper and draws directions to further extend our study.

II. PROBLEM STATEMENT AND RELEVANCE

Software defects are logic or implementation errors that cause the system to operate in unintended ways or to produce incorrect results. SDP consists in identifying the software components that contain defects.

Let us consider a software system $\mathcal{S}yst$ described by a set of software entities (modules, classes, methods or functions, depending on the chosen granularity), $\mathcal{S}yst = \{e_1, e_2, \ldots, e_n\}$. The software entities are represented as numerical vectors and are characterized by a set of software features (usually software metrics) $\mathcal{SF} = \{sf_1, sf_2, \ldots, sf_\ell\}$. Thus, each element from the vector associated with a software entity represents the value of a software feature (or metric) computed for that entity. A software entity $e_i \in \mathcal{S}yst$ is represented as an ℓ-dimensional vector, $e_i = (e_{i1}, e_{i2}, \ldots, e_{i\ell})$, where e_{ij} expresses the value of the software metric sf_j computed for the software entity e_i.

From the perspective of supervised machine learning, SDP can be formulated as a binary classification problem. There are two possible target classes for the software defect predictor (or classifier): the *positive* class of the defective software entities (labeled as "+," or "1"), and the *negative* class of the defect-free software entities (labeled as "-," or "0"). A training data set including both positive and negative samples will be used for building the software defect predictor that will be further used for classifying unseen instances (software entities) in order to predict their defect-proneness.

The target function to be learned in a SDP task is the mapping $t : \mathcal{S}yst \rightarrow \{\text{"+,"} \text{ "-"}\}$ which has to assign to each software entity e a class $t(e) \in \{\text{"+,"} \text{ "-"}\}$, denoting if the entity is defective or not. Thus, from a supervised classification perspective, the SDP task may be formalised as searching a hypothesis $h \approx t$ (i.e., an approximation of the target function to be learned) that best fits the training data.

SDP has a broad applicability. Clark and Zubrow [3] have analysed the importance of predicting software defects. One important motivation for performing defect prediction is that it helps software managers to measure how software projects evolve. In addition, it supports process management by assessing the software product's quality [3], thus being essential for effective software quality assurance. As shown in [5], SDP significantly reduces the cost of the processes that aim at ensuring the quality of software.

Software quality assurance involves numerous processes, including testing and code review, also called code inspection. SDP makes testing more efficient by allowing to focus on the components identified as defective [1]. By increasing the effectiveness of testing, SDP contributes to improving the quality of the next versions of a software project. Identifying software defects is also useful for guiding code review by indicating the locations in the source code that are very likely to be defective and thus require particular attention.

SDP is also useful for predicting software reliability, which is imperative in software development, particularly for large scale and complex software projects [4].

III. BACKGROUND

The current section starts by describing, in Section III-A, the publicly available data sets used as case studies for SDP. Section III-B reviews existing supervised machine learning-based solutions for SDP. The section ends with a description of the features used for SDP, both manually and automatically engineered ones.

The prediction of defects in software systems is a highly active research area. For instance, Hall *et al.* [7] have identified, in a systematic review of SDP, 208 studies on defect prediction, all published between 2000 and 2010, and numerous other studies have been published since then.

There is a great interest in developing new high-performance software defect predictors. Besides the interest in developing accurate and robust defect predictors, there is also interest in defining new relevant software features on the basis of which to distinguish between defective and non-defective software modules. Therefore, the research efforts in the field of SDP take one of the following two directions: proposing new accurate classifiers or designing new relevant features [11].

A. DATA SETS FOR SDP

The vast majority of existing studies [11]–[22], have considered, as experimental data, some of the SDP data sets available in Promise Software Engineering Repository [23], which is currently known as SeaCraft (*Software Engineering Artifacts Can Really Assist Future Tasks*) [24]. They contain static OO metrics (such as the CK metrics proposed by Chidamber and Kemerer [25]) or traditional metrics associated with the quality of the procedural source code (such as the ones proposed by McCabe [26]).

Other publicly available data sets for SDP are the following: Nasa [27], Eclipse [28], Softlab [29], ReLink [30], AEEEM [31], Netgene [32], JIT [33], MJ12A [34], Audi [35], Shippey [36], GitHub [37], Rnalytica [38], FJIT [39] and Unified [40].

As suggested in the software engineering literature, the publicly available and thus reused SDP data sets are subject to two problems: the noisy labels [10] and the fact that the software features are insufficient or insufficiently relevant [41]. The noisy labels negatively affect the SDP models, while also predisposing the results of experimental evaluations to be unreliable [10], while the lack of significant features considerably limits the SDP performance [41].

Regarding the problem of anomalous class labels, most publications focus on the SZZ algorithm [42], [43], which is the most used algorithm for collecting defect labels [10]. Therefore, the majority of the public SDP data sets, including Eclipse [28], AEEEM [31], Netgene [32], Shippey [36], GitHub [37], JIT [33], Audi [35] and FJIT [39] are labeled using SZZ.

More studies revealed multiple issues with SZZ, caused by identifying insignificant changes [44], disregarding the field mentioning the affected version from issue reports [45], using a six-month time frame for attributing defects to releases [38] or relying on the supposedly correct labeling of issues in the issue tracking system.

In a very recent such study, Herbond *et al.* [10] performed an empirical assessment, on 398 releases of 38 Apache projects, focused on the defect labeling effectuated by the SZZ algorithm. The study concluded that SZZ misses approximately one fifth of the bug fixing commits, while only about half of the commits identified as bug fixing commits were truly bug fixes.

The authors have also assessed SZZ-RA [46], which is the state-of-the-art variant of SZZ. The experimental results disprove the loss of bug fixing commits, but the problem that only about half of the commits detected as bug fixes are indeed bug fixes persists.

In order to mitigate these problems, Herbond *et al.* [10] have slightly extended SZZ-RA, by adding a filter to ignore documentation and test files and by linking commits and issues based on Jira issue pattern.

B. SUPERVISED LEARNING-BASED SDP APPROACHES

Predictive machine learning models have been extensively applied in the SDP literature with the goal of predicting software defects.

Ruchika Malhotra [47] has compared statistical and ML methods, as solutions for SDP. In particular, Logistic Regression has been compared with six ML approaches comprising Decision Trees, Artificial Neural Networks, Support Vector Machines, Cascade Correlation Networks, Group Method of Data Handling (GMDH) Polynomial Networks, and Gene Expression Programming. These learning models have been evaluated on two *Ar* data sets and the best performance has been obtained using Decision Trees.

Panichella *et al.* [48] proposed a combined approach called COmbined DEfect Predictor (CODEP), which combines the classifications provided by different ML techniques to improve the detection of defective entities. CODEP has been evaluated on ten open source software systems in the context of cross-project SDP. The authors concluded that the accuracy of the predictions has been improved by combining different classifiers.

Xuan *et al.* [15] investigated the performance of within-project defect prediction based on 10 defect data sets from the Promise repository using six state-of-the-art ML approaches. Ten-fold cross-validation has been performed based on each data set and several evaluation measures were reported.

In order to better cope with noise and imprecise information, Marian *et al.* [16] have investigated a fuzzy Decision Tree method for SDP. The experimental results obtained on JEdit and Ant demonstrated the superior performance of the fuzzy approach when compared to a non-fuzzy approach.

A solution for SDP using a Bayesian approach has been proposed by Okutan and Yildiz [20]. The authors have applied the K2 algorithm [49] on nine publicly available data sets. Two new software metrics have been added to the software metrics from the Promise repository: number of developers (NOD) and lack of coding quality (LOCQ). The efficiency of different software metric pairs has been comparatively analyzed.

Highly appealing in the SDP literature are the cross-project defect predictors. They allow predicting defects in a target software system based on historical data from other systems. Therefore, they are more general and allow predicting defects in projects with limited historical data.

The problem of cross-project SDP, which allows predicting defects in a target software system based on historical data from other systems, has been approached in several studies including the ones of Yu and Mishra [50], Jaechang and Sunghunin [51] or Canfora *et al.* [14].

C. FEATURES USED FOR SDP

The SDP literature comprises various approaches proposed to engineer features relevant for SDP (usually software metrics that are considered to be appropriate for discriminating between defects and non-defects) as well as methods to automatically learn features using machine learning techniques, particularly deep learning models.

Regarding the insufficiency of relevant features for enabling the discrimination between defective and defect-free software entities, a relatively recent but active research direction in the SDP literature aims at defining new software features that are relevant for SDP.

Along this direction, in the last two decades, a noteworthy amount of research studies focused on the reliance of coupling and cohesion for predicting software defects [52]. If until relatively recently the studies focused exclusively on the coupling and cohesion metrics from the traditional suites (such as the Chidamber and Kemerer [25] metrics suite), the latest studies are concerned with updating, extending and complementing them, by proposing new relevant coupling and cohesion measures [53], [54].

A systematic mapping study on object-oriented (OO) coupling and cohesion metrics has been performed by Tiwari and Rathore [52]. The authors selected 137 research papers. Of these, 17% introduced new coupling metrics, 8% introduced new cohesion metrics, while 24% introduced both coupling metrics and cohesion metrics. The rest of the studies (51%) focused only on assessing the existing metrics suites. The prevalent criterion by which the coupling and cohesion metrics have been evaluated is their relevance for predicting software defects.

In a subsequent study, Rathore and Kumar [55] have conducted a survey on the existing approaches for SDP, with emphasis on the considered software metrics, quality of data, prediction models and performance indicators. Their review uncovered that the majority of the studies (39%) used OO metrics. The explanation Rathore and Kumar have formulated for the high use of OO metrics for SDP is the inability of traditional software metrics to capture OO features that underlie the modern software development practices, including coupling and cohesion. The authors concluded that more studies concerned with the proposal and the assessment of new metrics suites are necessary.

An approach for automatically learning semantic features from token vectors extracted from Abstract Syntax Trees (ASTs) has been proposed by Wang *et al.* [11]. The authors have used Deep Belief Networks (DBNs) to automatically learn features from token vectors extracted from the programs ASTs. The features have then been used for both within-project and cross-project SDP. Ten open source projects from the Promise repository have been considered. The semantic features have been comparatively evaluated against 20 traditional features (software metrics in the Promise repository), as well as the term frequencies of the AST nodes (i.e., the ones used to train the DBNs). The evaluation results have confirmed that the semantic features are able to lead to superior predictive performance and thus are more relevant to SDP.

Features that have been automatically learned through a process similar to the one proposed by Wang *et al.* [11] are combined with traditional features in a subsequent study performed by Li *et al.* [17]. To generate semantic and syntactic features, DBN has been replaced by Li *et al.* with CNN, given that the Deep Learning community claims that CNN is better than DBN, since the latter can capture local patterns better than the former. The automatically learned features have been fed into a Logistic Regression classifier, which has been evaluated on 7 open source software projects from Promise. The empirical results confirmed that the CNN based prediction model outperforms the classifiers based on traditional features, while combining the automatically learned features with traditional features raises performance even more.

Another study proposing using AST-based features for SDP is the one performed by Dam *et al.* [18]. After highlighting that traditional software metrics are not so effective, while code tokens carry semantic information, the authors have proposed a tree-structured network of Long-Short Term Memory (LSTM) units as a SDP prediction model fed with AST embeddings. The features generated by LSTMs have been fed into traditional classifiers (Logistic Regression and Random Forest). As evaluation case studies, the authors considered the same ten open source Java projects from the Promise repository as in the study performed by Wang *et al.* [11], but they have also considered a data set from open source projects contributed by Samsung and developed in the C programming language. As empirical results, Random Forest performed better on the Samsung data set, while in the case of the Promise data sets, the Logistic Regression proved superior performance.

Huo *et al.* [19] have proposed *Convolutional Neural Network for Comments Augmented Programs* (CAP-CNN) as a model for SDP. Their approach is based on using pretrained Word2vec to encode code and comments into numeric vectors and then feeding the so-obtained vectors into two separate CNNs. Eight Promise data sets have been employed in the empirical evaluation, while using resampling for their balancing. The evaluation results highlighted that CAP-CNN outperformed, for most experiments, CNN, as well as, standard classifiers such as Logistic Regression or Naive Bayes, but also Deep Belief Network [11].

In a previous study [56], we have also proposed a semantic features based hybrid SDP model combining Artificial Neural Networks with Gradual Relational Association Rules (GRARs). After encoding the source code and comments into fixed-length numeric vectors, GRARs mining has been employed to uncover interesting GRARs that are able to discriminate between defective and defect-free software components. Based on the differentiating GRARs, a Multilayer Perceptron is trained in order to learn the classification function. The empirical evaluation has been performed on 3 software projects from the Promise repository. The experimental results revealed that considering semantic features instead of traditional metrics preponderantly leads to superior SDP performance.

In 2020, Wang *et al.* [57] extended their prior publication [11], by doubling the within-project SDP with cross-project SDP, proposing new techniques to process incomplete code, updating the performance assessment scenarios and performing new experiments on open-source commercial projects. The experimental results reconfirmed that the proposed DBN-based semantic features outperform traditional SDP features.

Very recently, Sikic *et al.* [58] have proposed DP-GCNN, a SDP model based on a Convolutional Graph Neural Network (GCNN), which is fed with AST data. The neural network architecture employed is specifically tailored for graph data.As experimental data, the authors have considered 7 SDP data sets from the Promise repository. The experimental results revealed that DP-GCNN's performance is superior to those of the traditional SPD models and comparable with those of the state-of-the-art AST-based SDP models, including [57].

There are also traditional metrics based Deep Learning approaches in the SDP literature. Two recent studies [59], [60] have proposed Siamese Deep Neural Networks for SDP. Unlike the previously reviewed papers, the study was performed on NASA data sets instead of Promise data sets.

IV. METHODOLOGY

This section introduces the methodology underlying our study on how the software features used in SDP impact the SDP performance. The Calcite data set used as case study and the SDP software features proposed in the literature for this data set are described in Section IV-A.

The pipeline proposed for our study consists of the following stages:

1) **Adding conceptual features.** The additional set of conceptual software features proposed for capturing the semantics of the source code in order to enlarge the original feature set for the Calcite data set (Section IV-A) is introduced and detailed in Section IV-B.
2) **Features relevance analysis.** An in-depth analysis on various subsets (described in Section IV-C) of the original features set extended with the conceptual features proposed at the previous stage is then conducted in Section V. An extensive study performed on sixteen versions of Calcite has the goal of determining, through a supervised learning-based analysis reinforced by an unsupervised one, the set of features that brings a statistically significant improvement on the performance of predicting software defects on Calcite data.
3) **Predictive models performance analysis.** The last stage (Section VI) consists in a study on the performance of various defect predictors on the Calcite data set using the most relevant feature set previously identified.

A. CASE STUDY

As a case study, we selected Apache Calcite, an open-source dynamic data management framework [61]. The 16 considered releases of Calcite software are: 1.0.0, 1.1.0, 1.2.9, 1.3.0, 1.4.0, 1.5.0, 1.6.0, 1.7.0, 1.8.0, 1.9.0, 1.10.0, 1.11.0, 1.12.0, 1.13.0, 1.14.0 and 1.15.0. We are making our data sets publicly available [62].

We started from the data provided by Herbond *et al.* [10], namely the values for 4189 software features for each software instance from Calcite and the defect labels produced by their extended version of SZZ, which is SZZ-RA [46].

The 4189 features characterizing the software entities from the Calcite data set [10] include:

- Static code metrics collected using mecoSHARK [63].
- Clone metrics computed using mecoSHARK [63].
- Metrics based on the warnings produced by the PMD static analysis tool [64].
- AST node counts computed using coastSHARK [63].
- The number of different types of changes [65] and refactorings [66] from the last six months, collected using changeSHARK and refSHARK [63], respectively.

TABLE 1. The feature sets used in our case study.

Feature set label	# of software metrics	Description
All	4189	The entire feature set containing all available software metrics.
SM	3813	Static code software metrics.
PMD	110	The subset of metrics extracted using the PMD static analysis tool [64].
D'Ambros	170	The subset Code churn metrics proposed by D'Ambros et al. [31]
AST	61	The subset of software metrics extracted from the AST.

TABLE 2. Total number of software instances, number of defective software instances and rate of defective instances for all calcite versions.

Version	Instances	Defective instances	Defective rate
Calcite 1.0.0	1075	178	0.166
Calcite 1.1.0	1103	113	0.102
Calcite 1.2.0	1108	126	0.114
Calcite 1.3.0	1115	112	0.100
Calcite 1.4.0	1127	123	0.109
Calcite 1.5.0	1176	103	0.088
Calcite 1.6.0	1193	107	0.090
Calcite 1.7.0	1252	128	0.102
Calcite 1.8.0	1301	101	0.078
Calcite 1.9.0	1310	90	0.069
Calcite 1.10.0	1310	84	0.064
Calcite 1.11.0	1331	80	0.060
Calcite 1.12.0	1415	81	0.057
Calcite 1.13.0	1275	53	0.042
Calcite 1.14.0	1308	53	0.041
Calcite 1.15.0	1352	45	0.033

TABLE 3. Difficulty values for the versions of calcite software, computed considering the entire set of features proposed in [10] (denoted by **All**).

Version	Difficulty "+"	Difficulty "-"	Overall difficulty
Calcite 1.0.0	0.596	0.111	0.192
Calcite 1.1.0	0.628	0.076	0.132
Calcite 1.2.0	0.643	0.067	0.133
Calcite 1.3.0	0.616	0.066	0.121
Calcite 1.4.0	0.642	0.060	0.123
Calcite 1.5.0	0.738	0.056	0.116
Calcite 1.6.0	0.710	0.051	0.110
Calcite 1.7.0	0.711	0.064	0.130
Calcite 1.8.0	0.723	0.054	0.106
Calcite 1.9.0	0.778	0.048	0.098
Calcite 1.10.0	0.762	0.052	0.098
Calcite 1.11.0	0.800	0.047	0.095
Calcite 1.12.0	0.790	0.032	0.076
Calcite 1.13.0	0.849	0.029	0.064
Calcite 1.14.0	0.887	0.028	0.063
Calcite 1.15.0	0.911	0.031	0.061

- Code churn metrics proposed by Moser *et al.* [67], Hassan [68] and D'Ambros *et al.* [31].

Additionally, all the 13 schemes proposed by Zhang *et al.* [69] for aggregating class, interface, enum, method, attribute and annotation metrics have been applied to expand the feature space.

We are focusing in our study on five features subsets: the entire set of software metrics and four other feature subsets with the largest dimensionality. The features sets considered in our case study are summarized in Table 1. Each row from the table indicates a feature (sub)set, its dimensionality and a brief description of the contained features.

Descriptive statistics for the available versions of Calcite are presented in Table 2. For all Calcite versions, the total number of software instances, number of defective software instances and defective rate are given. Table 2 reveals that both the defective rate and the number of faulty entities have a general decreasing tendency during the evolution of the Calcite software system. In the latest release of the software (version 1.15.0) there is the lowest defective rate and the smallest number of software defects. This tendency is expectable since as the system evolved it was improved and defects were corrected.

For better understanding the complexity of the software defect prediction task during the evolution of the Calcite software, we computed for each data set (corresponding to a software version) three **difficulty** measures. Following the definition given by Zhang *et al.* [70], the *difficulty* of a given class *c* ("+" or "-," in our case) is computed as the proportion of software entities labeled as *c* for which the nearest neighbor (computed using the Euclidean distance) belongs to the opposite class (i.e., "-" or "+," respectively). The overall difficulty of a data set is expressed as the weighted average of the difficulties computed for the defective (positive) and non-defective (negative) classes. Intuitively, the *difficulty* of a certain class indicates how hard is to distinguish the instances belonging to that class, considering a given vectorial representation for the software entities.

Table 3 presents the values for the previously described difficulty measures for each version of the Calcite system, considering the entire feature set (labeled as **All** in Table 1). The second column from the table denotes the difficulty for the defective class (the *positive* one), while the third column depicts the difficulty values for the non-defective class (the *negative* one).

Figure 1 plots the variation of the defective rates and difficulties for each version of the Calcite data set. One can observe from the figure that there is a strong correlation between the defective rate and the difficulty values during the evolution of the software. The same relationship may be observed from Table 4 that presents the Pearson correlation coefficients [71] between the defective rates and the difficulty values for all versions of the Calcite software. However, even if there is a strong linear relationship between the defective rate and the difficulty for the *positive* class, the correlation is inverse (negative) and thus it indicates that the number of defects and the difficulty for the defective class tend to move in opposite size and direction from one another. This is

TABLE 4. Pearson correlations between the defective rates and the difficulty values for all versions of calcite software.

	Difficulty "+"	Difficulty "-"	Overall difficulty
Defective rate	-0.919	0.955	0.985

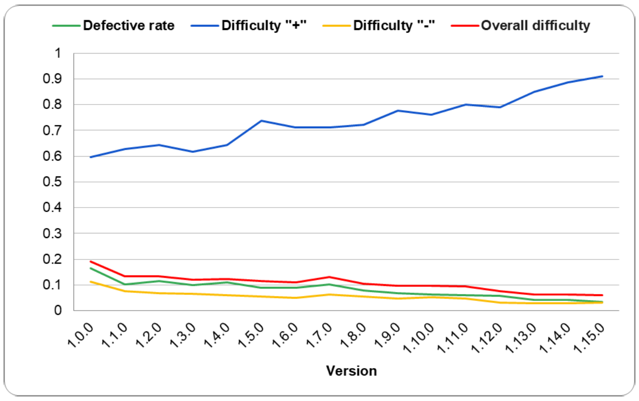


FIGURE 1. Variation of defective rates vs. difficulties during the evolution of the Calcite system.

not unexpected since, intuitively, the smaller the number of defects is, the harder it is to differentiate them from the entire set of entities.

Due to the severe imbalancement of the two classes (the number of defects are highly outnumbered by the number of non-defective ones), the main difficulty is that of predicting the *positive* class. Therefore, we consider, as the real difficulty, the one on the *positive* class. A difficulty of 1.0 means that every instance of the respective class has as its nearest neighbour an entry from the other class. That level of dissimilarity between *positive* entries makes it incredibly difficult for a classification model to correctly identify that class. It can be observed that some data sets have difficulties that come close to 1.0, while for all of them, the *positive* entries are mostly surrounded by negative ones (difficulty > 0.5).

B. PROPOSED CONCEPTUAL-BASED FEATURES

As shown in Section III, numerous ML techniques applied for predicting software defects are based on using classic software metrics as input features. Using these data sets, SDP models can be built without considering the source code of the analyzed software.

Rathore and Kumar [55] concluded their extensive review on existing SDP approaches by emphasizing the need to propose and validate new features that can be relevant for discriminating between defective and non-defective software entities.

Moreover, various studies from the literature [17], [18] reveal that the traditional software metrics are unable to capture the semantics of the source code. Besides the structural relationships existing in a software system and expressed by most of the software metrics, it would be relevant to consider the textual information contained in the source code as well. In this regard, it is agreed that *conceptual* software features extracted from the source code (identifiers, comments, etc) are able to capture semantic characteristics that structural metrics are not entirely able to express. Extracting conceptual (semantic) information from comments and identifiers within the source code has been also investigated in the software engineering literature for expressing *conceptual coupling* between software components [72], [73].

Since 2016, SDP many researches focused on using DL models and semantic features extracted from the source code. Recent research papers (Yang *et al.* [74] and Wang *et al.* [11]) introduced Deep Belief Networks (DBN) for performing defect prediction based on code analysis. Wang *et al.* [11] argued that besides the classical software metrics, the semantics of code should also be considered for SDP. The authors proposed DBN to automatically learn semantic features from input vectors of tokens extracted from the AST of the source code. Dam *et al.* [18] first used Long Short Term Memory (LSTM) networks to learn semantic features from the AST which were used to train a Logistic Regression (LR) and a Random Forest (RF) model. Traditional metrics were combined with features learnt from AST using a Convolutional Neural Network (CNN) by Li *et al.* [17]. Šikic *et al.* [58] used a graph convolutional neural network (GCNN) for processing the information of the nodes and edges from the AST of the source code for classifying the module as being defective or non-defective.

Doc2Vec [75] and LSI [76] models may also be used for unsupervisedly learning conceptual-based features from the source code. Both models are used for representing texts (in our case, source code) as fixed-length numerical vectors.

Doc2Vec, or Paragraph Vector is a multilayer perceptron (MLP) based model proposed by Le and Mikolov [75]. It allows expressing variable-length textual information as a fixed-length dense numeric vector, called *paragraph vector*, thus being an alternative to common models such as bag-of-words and bag-of-n-grams.

A first advantage of Doc2Vec over the traditional models is that it considers the semantic distance between words [75]. Therefore, *private* will be closer to *protected* than to *boolean*. An additional advantage over bag-of-words is that it also takes into consideration the words order, at least in a small context. Despite the fact that bag-of-n-grams, with a large n, also takes into account the word order in short contexts, it suffers from high dimensionality and data sparsity.

Doc2Vec extends Word2Vec, which learns distributed vector representations of words. Doc2Vec learns distributed representations for variable-length pieces of text, called paragraphs, ranging from sentences to entire documents.

The experimental results of previous studies we have conducted [54], [56] revealed that combining Doc2Vec and LSI is appropriate and increases the predictive performance.

Using Doc2Vec and LSI, the entities from a software system are represented as *conceptual vectors*. The conceptual vectors are vectors of numerical values corresponding to a set $\mathcal{S} = \{s_1, s_2, \ldots, s_\ell\}$ of conceptual (or semantic) features unsupervisedly learned from the source code. Thus, a software entity e_i is represented as an ℓ-dimensional vector in Doc2Vec and LSI spaces:

(1) $e_i^{Doc2Vec} = (e_{i1}^{Doc2Vec}, \cdots, e_{i\ell}^{Doc2Vec})$, where $e_{ij}^{Doc2Vec}$ $(\forall 1 \leq j \leq \ell)$ denotes the value of the j-th semantic feature computed for the entity e_i by using Doc2Vec.

(2) $e_i^{LSI} = (e_{i1}^{LSI}, \cdots, e_{i\ell}^{LSI})$, where e_{ij}^{LSI} $(\forall 1 \leq j \leq \ell)$ denotes the value of the j-th semantic feature computed for the entity e_i by using LSI.

In our study, for extracting the conceptual vectors corresponding to the software entities, the unsupervised learning models Doc2Vec and LSI are used. Both Doc2Vec [75] and LSI [76], also known as Latent Semantic Analysis (LSA), are models aimed to represent texts of variable lengths as fixed-length numeric vectors capturing semantic characteristics, but Doc2Vec is a prediction-based model trained using backpropagation together with the stochastic gradient descent, while LSI is a statistical, count-based model.

We opted for $\ell = 30$ as the length of the conceptual vectors extracted using Doc2Vec and LSI. For building the corpora for training, we filtered the source code (including comments) afferent to each class so as to keep only the tokens presumably carrying semantic meaning. So, operators, special symbols, English stop words or Java keywords have been eliminated. For both Doc2Vec and LSI, we have used the implementation offered by Gensim [77].

C. FEATURE SETS USED

In this section we are describing the feature sets that will be further used n Section V in our study performed on Calcite data set. The proposed study is aimed to determine, through a supervised learning-based analysis reinforced by an unsupervised one, the set of features that brings a statistically significant improvement of the SDP performance on Calcite data.

Twelve feature sets will be further experimented:

1.-5. The first five feature sets (labeled as **All**, **SM**, **PMD**, **D'Ambros**, **AST**) are the features (sub)sets described in Table 1.

6.-8. The next three feature sets, labeled as **AST+SM**, **AST+PMD** and **AST+D'Ambros** are obtained by fusing the AST-based features and the SM, PMD and D'Ambros features sets, respectively. We decided to use the AST-based features in all these combinations since the literature reveals various approaches [17], [18], [58] in which deep learning models are used to learn relevant features starting from the ASTs of the source code.

9.-10. The next two feature sets, denoted by **Doc2Vec** and **LSI**, are the conceptual features from Doc2Vec and LSI spaces, as described in Section IV-B.

11. The feature set labeled as **Doc2Vec+LSI** is represented by fusing the Doc2Vec and LSI features.

12. The last feature set, denoted by **All+Doc2Vec+LSI**, is obtained by fusing the feature set **All** with the conceptual features within the **Doc2Vec+LSI** feature set.

V. FEATURE SETS RELEVANCE ANALYSIS

As directions for further research in SDP, Herbond *et al.* [10] have recommended that analyses have to be performed in order to uncover the most relevant subsets of the extensive metrics set they proposed.

Following this idea and the methodology introduced in Section IV, with the goal of answering RQ2, we are examining the feature sets proposed in Section IV-C for deciding, through supervised and unsupervised learning-based analyses, their relevance in the context of SDP applied on the Calcite data set.

In Section V-A a supervised learning-based analysis will be conducted to decide the best feature set (from those described in Section IV-C), namely the set of features that provides a statistically significant performance improvement for a deep learning defect predictor applied on all the versions of the Calcite software. Afterwards, the results of the supervised learning-based analysis are strengthen in Section V-B by an unsupervised learning-based study.

A. SUPERVISED ANALYSIS

For determining which is the most relevant feature set for characterizing the software entities from the Calcite system (i.e., the set of features able to discriminate best between defective and non-defective entities) we decided to use a highly performant deep learning classifier and to evaluate its performance (in terms of multiple performance evaluation metrics) on all versions of Calcite, described by using all 12 feature sets (described in Section IV-C).

The deep learning classifier we decided to use, denoted by DL-FASTAI, is implemented in the FastAI machine learning library [78]. It is composed of an Artificial Neural Network combined with embeddings of the input layer. The architecture consists of 1 input, 1 output and 2-4 hidden layers, depending on the number of features. Compared to other deep learning models, especially Convolutional Neural Networks (CNNs) used in computer vision, this ANN model is very small and fast, with training times under 2 minutes on our data set and inference time under 1 second per instance at runtime, making it suitable for real-time use. The model is trained using the FastAI 'fit one cycle' method, which uses a learning rate that varies according to a specific pattern: first it increases, then it decreases and the process is repeated for each epoch.

In order to evaluate the performance of the DL-FASTAI model, we employed the following evaluation methodology. The data was split into 70% train, 10% validation and 20% test sets. In order to get consistent results, cross-validation on 10 experiments with different splits had been done.

During the cross-validation process, the confusion matrix for the binary classification task has been computed for each testing subset. Based on the values from the confusion matrix

(TP - number of true positives, FP - number of false positives, TN - number of true negatives and FN - number of false negatives), multiple performance metrics from the supervised learning literature have been computed. For each metric, the values have been averaged during the cross-validation process and the 95% confidence interval (CI) of the mean values has been calculated. The performance metrics used in our evaluations are summarized below:

1) **Accuracy** (Acc, computed as the percentage of correct classifications), $Acc = \frac{TP+TN}{TP+FP+TN+FN}$.
2) **Precision for the positive class** ($Prec$, also known as *positive predictive value*), $Prec = \frac{TP}{TP+FP}$.
3) **Precision for the negative class** (NPV, the *negative predictive value*), $NPV = \frac{TN}{TN+FN}$.
4) **Sensitivity** ($Sens$, the true positive rate, also known as *recall*), computed as $Sens = \frac{TP}{TP+FN}$.
5) **Specificity** ($Spec$, the true negative rate), $Spec = \frac{TN}{TN+FP}$.
6) **Area under the ROC curve** (AUC). The SDP literature reveals that AUC is a suitable measure for evaluating the performance of the software defect classifiers [79]. In general, the AUC measure is employed for approaches that yield a single value, which is then converted into a class label using a threshold. Thus, for each threshold value, the point $(1 - Spec, Sens)$ is represented on a plot and the AUC is computed as the area under this curve. For the approaches where the direct output of the defect classifier is the class label, there is only one $(1 - Spec, Sens)$ point, which is linked to the $(0, 0)$ and $(1, 1)$ points. The AUC measure represents the area under the trapezoid and is computed as $AUC = \frac{Sens+Spec}{2}$.
7) **Area under the Precision-Recall curve** ($AUPRC$). Somehow similarly to the ROC curve, the Precision-Recall curve represents a two-dimensional plot of (*sensitivity*, *precision*) points computed for different values for the threshold applied for deciding the output class. For the classifiers for which the output is the class label (obtained without thresholding the output value), the point (*sensitivity*, *precision*) is linked to the points at (0,1) and (1,0), and the area under the resulting trapezoid is computed as $AUPRC = \frac{(Prec+Sens)}{2}$. $AUPRC$ is considered a good measure for imbalanced classification and it also has higher values for better classifiers.
8) **Matthews Correlation Coefficient** (MCC) [80] is also considered to be a good evaluation metric for imbalanced data sets and is computed as $MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$. MCC takes values in the [-1, 1] interval, the value 1 denoting a perfect classifier. The value -1 is returned if every prediction of the model is incorrect (TP = TN = 0). A value of 0 means the same performance as random guessing. Also, if any row or column of the confusion matrix is 0, MCC is undefined.

TABLE 5. The winning feature set(s), $win(v)$, and the number of performance metrics (denoted by $n(v, win(v))$) whose values are the highest for the winning feature set(s), computed for each calcite version.

Version	Winning feature sets ($win(v)$)	# of performance metrics $n(v, win(v))$
Calcite 1.0.0	Doc2Vec+LSI	11
Calcite 1.1.0	All+Doc2Vec+LSI	9
Calcite 1.2.0	All+Doc2Vec+LSI, d'Ambros	5
Calcite 1.3.0	LSI	7
Calcite 1.4.0	Doc2Vec+LSI	7
Calcite 1.5.0	All+Doc2Vec+LSI	6
Calcite 1.6.0	AST+D'Ambros, LSI	5
Calcite 1.7.0	LSI	7
Calcite 1.8.0	Doc2Vec	6
Calcite 1.9.0	Doc2Vec+LSI	7
Calcite 1.10.0	Doc2Vec+LSI	7
Calcite 1.11.0	Doc2Vec	6
Calcite 1.12.0	d'Ambros	5
Calcite 1.13.0	Doc2Vec+LSI	8
Calcite 1.14.0	Doc2Vec+LSI	5
Calcite 1.15.0	AST+D'Ambros	5

9) **F-score for the "+" class** ($F\text{-}score^+$, computed as the harmonic mean between the precision for the positive class and the sensitivity/recall), $F\text{-}score^+ = \frac{2}{\frac{1}{Prec}+\frac{1}{Sens}}$.
10) **F-score for the "-" class** ($F\text{-}score^-$, computed as the harmonic mean between the precision for the negative class - NPV and the specificity - recall of the negative class), $F\text{-}score^- = \frac{2}{\frac{1}{NPV}+\frac{1}{Spec}}$.
11) **Overall F-score** ($F1$) computed as the average between $F\text{-}score^+$ and $F\text{-}score^-$.
12) **Weighted F-score** ($F1_w$) is computed as the weighted average between $F\text{-}score^+$ and $F\text{-}score^-$, where the weights are computed as the defective and non-defective rates, respectively.

All the previously mentioned evaluation measures range from 0 to 1, excepting MCC, which ranges from -1 to 1. For better classifiers, larger values are expected.

For all 16 versions of the Calcite system and all 12 feature sets selected for analysis (as presented in Section IV-C), the 12 evaluation metrics previously described have been computed. For a given version v ($v \in [1.0.0 - 1.15.0]$) of the Calcite framework and a given feature set fs ($fs \in$ {All, SM, PMD, D'Ambros, AST, AST+SM, AST+PMD and AST+D'Ambros, Doc2Vec, LSI, Doc2Vec+LSI, All+Doc2Vec+LSI}), the value $n(v, fs)$ is computed as the number of performance metrics p ($p \in \{Acc, Prec, NPV, Sens, Spec, AUC, AUPRC, MCC, F\text{-}score^+, F - score^-, F1, F1_w\}$ whose values are the highest for the feature set fs. Next, the "winning" feature set for a version v, denoted by $win(v)$ is considered to be the one that maximizes $n(v, fs)$, i.e., $win(v) = \arg\max_{fs} n(v, fs)$. Then, for each feature set fs we compute the value $WIN(fs) = \sum_{v} win(v)$.

Table 5 presents, for each Calcite version v, the winning feature set(s), $win(v)$, and the number of performance metrics, $n(v, win(v))$, whose values are the highest for the winning feature set(s).

TABLE 6. Number of *WIN* values computed for each feature set, after evaluating the DL-FASTAI classifier on all calcite versions. The feature sets are listed in the decreasing order of the *WIN* values.

Feature set (fs)	$WIN(fs)$
Doc2Vec+LSI	6
All+Doc2Vec+LSI	3
LSI	3
AST+D'Ambros	2
Doc2Vec	2
D'Ambros	2

Table 6 presents the feature sets *fs* for which a non-zero value has been obtained for the *WIN*(*fs*) measure. The feature sets are listed in the decreasing order of the *WIN* values.

From Table 6 we observe that the feature set with the maximum number of wins is **Doc2Vec+LSI**, the feature set obtained by fusing the proposed Doc2Vec and LSI semantic features. We remark that the feature set containing only the original features (**All**) [10] was not the winning feature set for none of the Calcite versions. Still, the joint feature set All+Doc2Vec+LSI was the second best feature set (the winning feature set for 3 Calcite versions). This suggest that the conceptual features extracted from the source code through Doc2Vec and LSI are the best for distinguishing between the defective and non-defective software entities.

Tables 7 and 8 present the performance metrics values obtained by evaluating the DL-FASTAI classifier on all Calcite versions characterized by the Doc2Vec+LSI, All+Doc2Vec+LSI and All feature sets. 95% CIs are used for the results. For each of the Calcite versions, the feature set that provides the best performance metrics (the maximum number of best performance values) is highlighted.

From Tables 7 and 8 one observes that the **Doc2Vec+LSI** feature set is the best for 67% of the Calcite versions (10 out of 15), when compared to All+Doc2Vec+LSI and All feature sets. For verifying the statistical significance of the differences observed between the evaluation metrics values obtained for Doc2Vec+LSI features and All+Doc2Vec+LSI/All features, a one tailed paired Wilcoxon signed-rank test [81], [82] has been applied. The sample of values representing the performance metrics values obtained by the DL-FASTAI classifier for the Calcite versions and Doc2Vec+LSI feature set was tested against the samples of values obtained for All+Doc2Vec+LSI and All features, respectively. The obtained *p*-values of 0.0037779 (for Doc2Vec+LSI vs. All+Doc2Vec+LSI) and 0.000309 (for Doc2Vec+LSI vs. All) confirm a statistically significant improvement achieved by the Doc2Vec+LSI feature set, at a significance level of $\alpha = 0.01$.

The superiority of the Doc2Vec+LSI feature set with respect to All+Doc2Vec+LSI and All features is strongly correlated with the overall difficulty values, as shown in Figure 2. The figure plots the overall difficulty values computed for all Calcite versions and the feature sets Doc2Vec+LSI, All+Doc2Vec+LSI and All. A statistically significant difference, at a significance level of $\alpha = 0.01$, was observed between the difficulties obtained for the

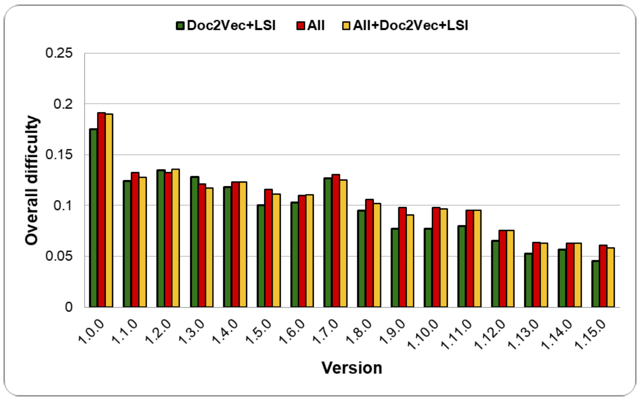


FIGURE 2. Overall difficulty values computed for all Calcite versions and the feature sets Doc2Vec+LSI, All+Doc2Vec+LSI and All.

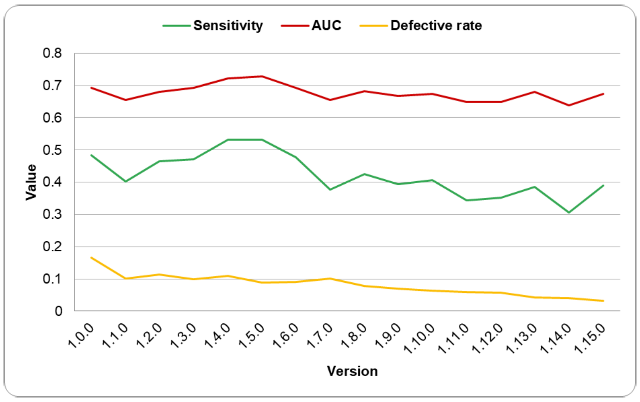


FIGURE 3. Variation of *sensitivity*, *AUC* and defective rates for the Calcite versions.

Doc2Vec+LSI features and the difficulties for the other feature sets (All+Doc2Vec+LSI and All) as provided by a one-tailed paired Wilcoxon signed-rank test: *p*-values of 0.000876 (between Doc2Vec+LSI and All) and 0.0024120 (between Doc2Vec+LSI and All+Doc2Vec+LSI).

Using the values from Tables 7 and 8 for the Doc2Vec+LSI feature set, we computed the Pearson correlation coefficients between the sample of defective rates for all versions of the system and the values obtained for sensitivity (*Sens*) and *AUC*. A strong correlation (0.66) has been observed between the sensitivity values and the defective rates and a moderate correlation (0.42) between the AUC values and the defective rates. Figure 3 depicts the variation of *sensitivity*, *AUC* and defective rates for the Calcite versions. A higher strength of the association between sensitivity values obtained by the DL-FASTAI classifier and the defective rates for the Calcite versions is expected. As shown in Figure 1, if the defective rate increases there is a general decrease of the difficulty for the "+" class and thus it is easier to recognize the defects, i.e. it is very likely that the DL-FASTAI classifier will obtain a higher true positive rate (sensitivity).

B. UNSUPERVISED ANALYSIS

In order to strengthen the supervised learning-based analysis performed in Section V-A and to better highlight that

TABLE 7. Performance metrics obtained by evaluating the DL-FASTAI classifier on calcite versions 1.0.0-1.7.0 and the feature sets **Doc2Vec+LSI**, **All+Doc2Vec+LSI** and **All**. 95% CI are used for the results.

Version	Feature set	Acc	$Prec$	NPV	$Sens$	$Spec$	AUC	$AUPRC$	MCC	$F\text{-}score^{+}$	$F\text{-}score^{-}$	$F1$	$F1_w$
1.0.0	Doc2Vec+LSI	0.834 ± 0.01	0.473 ± 0.04	0.903 ± 0.02	0.484 ± 0.08	0.901 ± 0.01	0.692 ± 0.04	0.478 ± 0.05	0.379 ± 0.06	0.472 ± 0.06	0.902 ± 0.01	0.687 ± 0.03	0.834 ± 0.01
	All+Doc2Vec+LSI	0.814 ± 0.02	0.430 ± 0.06	0.895 ± 0.01	0.449 ± 0.06	0.884 ± 0.02	0.666 ± 0.03	0.439 ± 0.05	0.327 ± 0.07	0.434 ± 0.05	0.889 ± 0.02	0.661 ± 0.03	0.817 ± 0.02
	All	0.821 ± 0.01	0.429 ± 0.04	0.894 ± 0.01	0.430 ± 0.08	0.895 ± 0.01	0.662 ± 0.04	0.429 ± 0.06	0.322 ± 0.06	0.424 ± 0.06	0.894 ± 0.01	0.659 ± 0.03	0.820 ± 0.01
1.1.0	Doc2Vec+LSI	0.854 ± 0.02	0.344 ± 0.08	0.928 ± 0.02	0.402 ± 0.08	0.908 ± 0.02	0.655 ± 0.04	0.373 ± 0.06	0.286 ± 0.07	0.357 ± 0.07	0.917 ± 0.01	0.637 ± 0.04	0.860 ± 0.02
	All+Doc2Vec+LSI	0.888 ± 0.02	0.472 ± 0.08	0.932 ± 0.01	0.410 ± 0.06	0.944 ± 0.02	0.677 ± 0.03	0.441 ± 0.05	0.373 ± 0.05	0.424 ± 0.04	0.937 ± 0.01	0.681 ± 0.02	0.885 ± 0.02
	All	0.875 ± 0.01	0.395 ± 0.06	0.930 ± 0.01	0.391 ± 0.07	0.932 ± 0.02	0.662 ± 0.03	0.393 ± 0.05	0.320 ± 0.05	0.380 ± 0.05	0.930 ± 0.01	0.655 ± 0.03	0.874 0.01
1.2.0	Doc2Vec+LSI	0.845 ±0.01	0.387 ± 0.06	0.927 ± 0.01	0.466 ± 0.06	0.896 ± 0.02	0.681 ± 0.02	0.427 ± 0.03	0.333 ± 0.04	0.409 ± 0.04	0.910 ± 0.01	0.660 ± 0.02	0.852 ± 0.01
	All+Doc2Vec+LSI	0.867 ±0.02	0.445 ± 0.06	0.923 ± 0.01	0.421 ± 0.07	0.926 ± 0.02	0.673 ± 0.03	0.433 ± 0.05	0.354 ± 0.05	0.423 ± 0.05	0.924 ± 0.01	0.673 ± 0.03	0.866 ± 0.02
	All	0.866 ±0.02	0.423 ± 0.06	0.920 ± 0.01	0.382 ± 0.08	0.929 ± 0.02	0.655 ± 0.04	0.403 ± 0.06	0.325 ± 0.07	0.397 ± 0.06	0.924 ± 0.01	0.660 ± 0.03	0.863 ± 0.02
1.3.0	Doc2Vec+LSI	0.870 ±0.02	0.384 ± 0.06	0.942 ± 0.01	0.472 ± 0.09	0.912 ± 0.03	0.692 ± 0.04	0.428 ± 0.06	0.350 ± 0.06	0.410 ± 0.05	0.926 ± 0.01	0.668 ± 0.03	0.876 ± 0.02
	All+Doc2Vec+LSI	0.894 ±0.01	0.451 ± 0.09	0.932 ± 0.01	0.355 ± 0.06	0.952 ± 0.01	0.653 ± 0.03	0.403 ± 0.07	0.341 ± 0.07	0.392 ± 0.07	0.942 ± 0.01	0.667 ± 0.03	0.889 ± 0.01
	All	0.881 ±0.02	0.407 ± 0.10	0.932 ± 0.01	0.376 ± 0.07	0.937 ± 0.02	0.656 ± 0.03	0.392 ± 0.07	0.322 ± 0.07	0.380 ± 0.07	0.934 ± 0.01	0.657 ± 0.04	0.880 ± 0.02
1.4.0	Doc2Vec+LSI	0.868 ±0.02	0.441 ± 0.06	0.937 ± 0.01	0.531 ± 0.09	0.912 ± 0.02	0.722 ± 0.04	0.486 ± 0.05	0.406 ± 0.05	0.470 ± 0.05	0.924 ± 0.01	0.697 ± 0.03	0.873 ± 0.02
	All+Doc2Vec+LSI	0.880 ±0.01	0.473 ± 0.10	0.917 ± 0.01	0.329 ± 0.07	0.951 ± 0.01	0.640 ± 0.03	0.401 ± 0.07	0.325 ± 0.06	0.375 ± 0.06	0.933 ± 0.01	0.654 ± 0.03	0.871 ± 0.01
	All	0.883 ±0.02	0.489 ± 0.05	0.921 ± 0.01	0.372 ± 0.06	0.949 ± 0.01	0.660 ± 0.03	0.430 ± 0.04	0.360 ± 0.04	0.414 ± 0.04	0.935 ± 0.01	0.674 ± 0.02	0.876 ± 0.02
1.5.0	Doc2Vec+LSI	0.892 ±0.02	0.393 ± 0.06	0.956 ± 0.01	0.531 ± 0.09	0.926 ± 0.01	0.728 ± 0.05	0.462 ± 0.07	0.397 ± 0.07	0.447 ± 0.06	0.940 ± 0.01	0.694 ± 0.04	0.899 ± 0.01
	All+Doc2Vec+LSI	0.913 ±0.01	0.479 ± 0.05	0.947 ± 0.01	0.420 ± 0.06	0.958 ± 0.01	0.689 ± 0.03	0.450 ± 0.04	0.399 ± 0.04	0.441 ±0.04	0.953 ± 0.01	0.697 ± 0.02	0.910 ± 0.01
	All	0.905 ±0.02	0.440 ± 0.09	0.945 ± 0.01	0.405 ± 0.08	0.951 ± 0.01	0.678 ± 0.04	0.422 ± 0.07	0.368 ± 0.08	0.414 ± 0.07	0.948 ± 0.01	0.681 ± 0.04	0.904 ± 0.02
1.6.0	Doc2Vec+LSI	0.873 ±0.02	0.348 ± 0.05	0.948 ± 0.01	0.477 ± 0.09	0.911 ± 0.02	0.694 ± 0.04	0.412 ± 0.05	0.336 ± 0.06	0.394 ± 0.05	0.929 ±0.01	0.661 ±0.03	0.882 ± 0.01
	All+Doc2Vec+LSI	0.901 ±0.01	0.439 ± 0.07	0.941 ± 0.01	0.384 ± 0.06	0.950 ± 0.01	0.667 ± 0.03	0.411 ± 0.05	0.353 ± 0.05	0.401 ± 0.05	0.946 ± 0.01	0.673 ± 0.03	0.898 ± 0.01
	All	0.896 ±0.02	0.416 ± 0.07	0.937 ± 0.01	0.333 ± 0.06	0.949 ± 0.02	0.641 ± 0.03	0.374 ± 0.05	0.312 ± 0.05	0.359 ± 0.05	0.943 ± 0.01	0.651 ± 0.03	0.892 ± 0.01
1.7.0	Doc2Vec+LSI	0.867 ±0.02	0.441 ± 0.09	0.917 ± 0.01	0.377 ± 0.08	0.934 ± 0.02	0.655 ± 0.04	0.409 ± 0.07	0.331 ± 0.07	0.398 ± 0.07	0.925 ± 0.01	0.661 ± 0.04	0.862 ± 0.02
	All+Doc2Vec+LSI	0.862 ±0.01	0.407 ± 0.04	0.914 ± 0.01	0.351 ± 0.07	0.932 ± 0.01	0.641 ± 0.03	0.379 ± 0.05	0.299 ± 0.05	0.370 ± 0.05	0.922 ± 0.01	0.646 ± 0.02	0.857 ± 0.01
	All	0.875 ±0.01	0.454 ± 0.07	0.908 ± 0.01	0.291 ± 0.07	0.954 ± 0.01	0.623 ± 0.04	0.373 ± 0.07	0.295 ± 0.07	0.346 ± 0.07	0.930 ± 0.01	0.638 ± 0.04	0.861 ± 0.02

Doc2Vec+LSI feature set is superior to the feature set (denoted by All in our study) proposed in the literature [10] in terms of differentiating between defective and non-defective software components we applied *t-distributed Stochastic Neighbor Embedding* (t-SNE) [83].

t-SNE is an unsupervised non-linear technique used for dimensionality reduction and feature extraction, as well as for visualizing and exploring high-dimensional data. It primarily focuses on retaining the local structure of the data, but also considers preserving its global structure. It works by finding similarities between the data points, showing similar points to be close to each other on the visual representations. The algorithm was implemented using the scikit-learn library [84].

Three versions of the Calcite framework have been selected for the unsupervised analysis, more specifically the ones with the maximum (version 1.0.0: overall difficulty 0.192), minimum (version 1.15.0: overall difficulty 0.061) and median (version 1.8.0: overall difficulty 0.106) overall difficulties. Figures 4, 5 and 6 illustrate the 2D t-SNE visualizations for the Calcite versions 1.0.0, 1.8.0 and 1.15.0 when using Doc2Vec+LSI and All feature sets for representing the software entities.

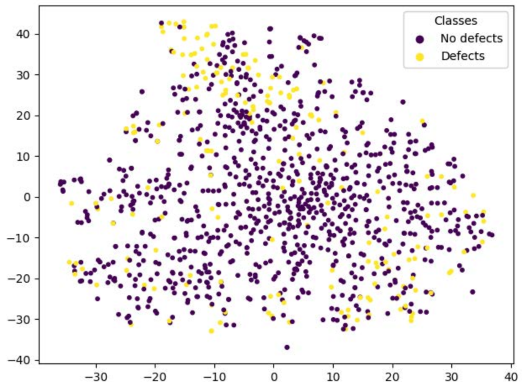
The plots from Figures 4, 5 and 6 reveal what the positive difficulty already predicted: the defective instances are very different from each other, usually similar to some non-defective ones. This behaviour is accentuated as the data set is more imbalanced, resulting in an almost uniform distribution of the positive instances in Figure 6. Furthermore, for the All feature set, small heterogeneous clusters are formed, making it even harder for a classifier to distinguish between positive and negative instances, due to their very high similarity. This unsupervised analysis supports the supervised one: $F\text{-}score^{+}$ is higher for version 1.0.0 than 1.15.0 and the metrics in general are better for the Doc2Vec+LSI feature set than the All one.

VI. PREDICTIVE MODELS PERFORMANCE ANALYSIS

Following the methodology introduced in Section IV, this section presents the last stage of our study. More specifically, we are going to comparatively analyse the performance of various defect predictors (DL-FASTAI, XGBoost,

TABLE 8. Performance metrics obtained by evaluating the DL-FASTAI classifier on Calcite versions 1.8.0-1.15.0 and the feature sets Doc2Vec+LSI, All+Doc2Vec+LSI and All. 95% CI are used for the results.

Version	Feature set	Acc	$Prec$	NPV	$Sens$	$Spec$	AUC	$AUPRC$	MCC	$F\text{-}score^+$	$F\text{-}score^-$	$F1$	$F1_w$
1.8.0	Doc2Vec+LSI	0.898 ±0.01	0.397 ± 0.09	0.951 ± 0.01	0.426 ± 0.11	0.939 ± 0.02	0.683 ± 0.05	0.412 ± 0.06	0.344 ± 0.06	0.378 ± 0.05	0.944 ± 0.01	0.661 ± 0.03	0.901 ± 0.01
	All+Doc2Vec+LSI	0.908 ±0.01	0.425 ± 0.11	0.942 ± 0.01	0.295 ± 0.05	0.960 ± 0.01	0.628 ± 0.02	0.360 ± 0.05	0.295 ± 0.05	0.323 ± 0.04	0.950 ± 0.01	0.637 ± 0.02	0.902 ± 0.01
	All	0.903 ±0.01	0.370 ± 0.05	0.946 ± 0.01	0.355 ± 0.06	0.949 ± 0.01	0.652 ± 0.03	0.363 ± 0.05	0.308 ± 0.05	0.357 ± 0.05	0.948 ± 0.01	0.652 ± 0.02	0.902 ± 0.01
1.9.0	Doc2Vec+LSI	0.812 ±0.18	0.330 ± 0.10	0.953 ± 0.01	0.393 ± 0.11	0.944 ± 0.02	0.668 ± 0.05	0.361 ± 0.09	0.308 ± 0.09	0.349 ± 0.09	0.948 ± 0.01	0.649 ± 0.04	0.906 ± 0.01
	All+Doc2Vec+LSI	0.798 ±0.17	0.243 ± 0.07	0.944 ± 0.01	0.278 ± 0.11	0.935 ± 0.03	0.606 ± 0.04	0.260 ± 0.08	0.196 ± 0.07	0.247 ± 0.07	0.939 ± 0.01	0.593 ± 0.04	0.890 ± 0.02
	All	0.881 ±0.02	0.240 ± 0.04	0.946 ± 0.01	0.315 ± 0.11	0.924 ± 0.02	0.619 ± 0.05	0.277 ± 0.06	0.206 ± 0.06	0.256 ± 0.05	0.935 ± 0.01	0.595 ± 0.03	0.886 ± 0.01
1.10.0	Doc2Vec+LSI	0.907 ±0.02	0.367 ± 0.10	0.958 ± 0.00	0.406 ± 0.06	0.942 ± 0.02	0.674 ± 0.03	0.387 ± 0.07	0.332 ± 0.07	0.375 ± 0.06	0.950 ± 0.01	0.662 ± 0.04	0.912 ± 0.01
	All+Doc2Vec+LSI	0.916 ±0.01	0.372 ± 0.11	0.952 ± 0.01	0.321 ± 0.07	0.957 ± 0.01	0.639 ± 0.03	0.346 ± 0.08	0.297 ± 0.08	0.336 ± 0.07	0.955 ± 0.01	0.646 ± 0.04	0.914 ± 0.01
	All	0.903 ±0.02	0.293 ± 0.07	0.950 ± 0.01	0.297 ± 0.10	0.946 ± 0.02	0.621 ± 0.04	0.295 ± 0.07	0.237 ± 0.07	0.278 ± 0.06	0.947 ± 0.01	0.613 ± 0.03	0.903 ± 0.01
1.11.0	Doc2Vec+LSI	0.915 ±0.01	0.343 ± 0.07	0.957 ± 0.01	0.344 ± 0.12	0.953 ± 0.02	0.649 ± 0.05	0.344 ± 0.06	0.286 ± 0.05	0.311 ± 0.04	0.955 ± 0.01	0.633 ± 0.02	0.916 ± 0.01
	All+Doc2Vec+LSI	0.919 ±0.01	0.288 ± 0.09	0.953 ± 0.01	0.273 ± 0.10	0.961 ± 0.01	0.617 ± 0.05	0.280 ± 0.09	0.236 ± 0.10	0.277 ± 0.09	0.957 ± 0.01	0.617 ± 0.05	0.916 ± 0.01
	All	0.907 ±0.01	0.265 ± 0.08	0.956 ± 0.01	0.315 ± 0.11	0.946 ± 0.02	0.630 ± 0.05	0.290 ± 0.08	0.235 ± 0.08	0.275 ± 0.08	0.950 ± 0.01	0.613 ± 0.04	0.910 ± 0.01
1.12.0	Doc2Vec+LSI	0.915 ±0.02	0.287 ± 0.09	0.965 ± 0.01	0.353 ± 0.07	0.945 ± 0.02	0.649 ± 0.03	0.320 ± 0.07	0.270 ± 0.08	0.305 ± 0.08	0.955 ± 0.01	0.630 ± 0.04	0.922 ± 0.01
	All+Doc2Vec+LSI	0.923 ±0.01	0.299 ± 0.07	0.967 ± 0.01	0.396 ± 0.11	0.952 ± 0.01	0.674 ± 0.05	0.347 ± 0.08	0.299 ± 0.08	0.329 ± 0.08	0.959 ± 0.01	0.644 ± 0.04	0.927 ± 0.01
	All	0.931 ±0.01	0.297 ± 0.12	0.960 ± 0.01	0.256 ± 0.11	0.968 ± 0.01	0.612 ± 0.05	0.276 ± 0.10	0.234 ± 0.10	0.262 ± 0.09	0.964 ± 0.01	0.613 ± 0.05	0.928 ± 0.01
1.13.0	Doc2Vec+LSI	0.951 ±0.01	0.475 ± 0.15	0.973 ± 0.01	0.386 ± 0.11	0.975 ± 0.01	0.681 ± 0.06	0.431 ± 0.11	0.390 ± 0.10	0.399 ± 0.09	0.974 ±0.01	0.687 ± 0.05	0.950 ± 0.01
	All+Doc2Vec+LSI	0.942 ±0.01	0.363 ± 0.07	0.976 ± 0.01	0.475 ± 0.13	0.964 ± 0.01	0.720 ±0.06	0.419 ± 0.08	0.377 ± 0.08	0.393 ± 0.07	0.970 ± 0.00	0.681 ± 0.04	0.945 ± 0.01
	All	0.848 ±0.18	0.345 ± 0.11	0.976 ± 0.01	0.451 ± 0.13	0.962 ± 0.01	0.707 ± 0.06	0.398 ± 0.11	0.359 ± 0.11	0.379 ± 0.10	0.969 ± 0.01	0.674 ± 0.05	0.944 ± 0.01
1.14.0	Doc2Vec+LSI	0.942 ±0.02	0.361 ± 0.11	0.969 ± 0.01	0.305 ± 0.08	0.970 ± 0.02	0.638 ± 0.04	0.333 ± 0.08	0.295 ± 0.08	0.314 ± 0.07	0.969 ± 0.01	0.642 ± 0.04	0.942 ± 0.01
	All+Doc2Vec+LSI	0.935 ±0.01	0.265 ± 0.07	0.968 ± 0.00	0.275 ± 0.07	0.964 ± 0.01	0.619 ± 0.03	0.270 ± 0.07	0.235 ± 0.07	0.267 ± 0.07	0.966 ± 0.01	0.616 ± 0.03	0.936 ± 0.01
	All	0.937 ±0.01	0.310 ± 0.07	0.970 ± 0.00	0.327 ± 0.10	0.964 ± 0.01	0.645 ± 0.04	0.318 ± 0.06	0.277 ± 0.06	0.299 ± 0.05	0.967 ± 0.01	0.633 ± 0.03	0.938 ± 0.01
1.15.0	Doc2Vec+LSI	0.941 ±0.01	0.274 ± 0.05	0.978 ± 0.01	0.390 ± 0.11	0.961 ± 0.01	0.675 ± 0.05	0.332 ± 0.06	0.291 ± 0.06	0.308 ± 0.06	0.969 ± 0.01	0.638 ± 0.03	0.947 ± 0.01
	All+Doc2Vec+LSI	0.942 ±0.01	0.258 ± 0.07	0.976 ± 0.00	0.329 ± 0.12	0.963 ± 0.02	0.646 ± 0.05	0.294 ± 0.07	0.251 ± 0.07	0.265 ± 0.07	0.969 ± 0.01	0.617 ± 0.04	0.945 ± 0.01
	All	0.954 ±0.01	0.387 ± 0.17	0.974 ± 0.00	0.245 ± 0.06	0.979 ± 0.01	0.612 ± 0.03	0.316 ± 0.10	0.272 ± 0.09	0.276 ± 0.07	0.976 ± 0.01	0.626 ± 0.04	0.953 ± 0.01



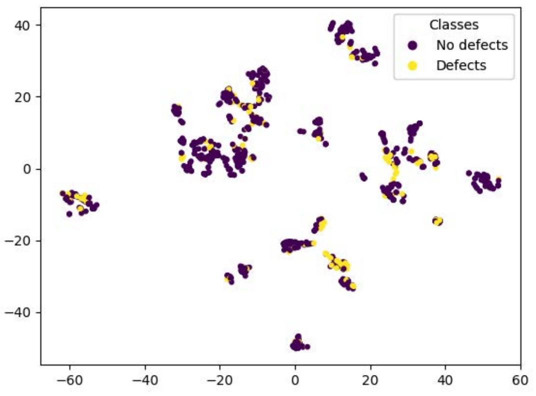
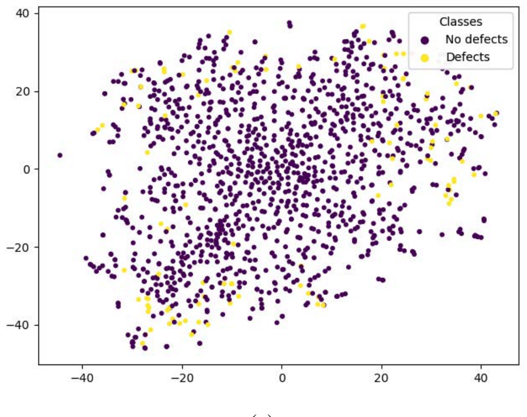
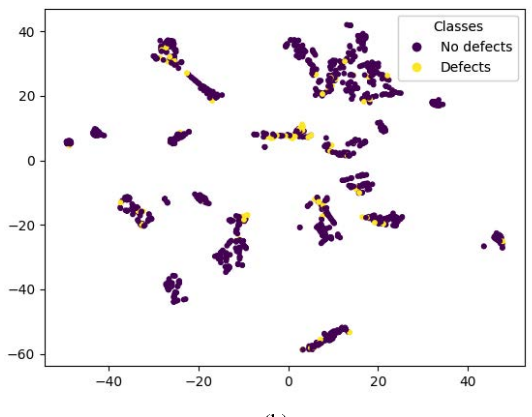


FIGURE 4. 2D t-SNE visualization for Calcite 1.0.0 for (a) Doc2Vec+LSI and (b) All feature sets.

SVM, ANN) on the versions of Calcite characterized by the most relevant feature set (Doc2Vec+LSI) identified through the study conducted in Section V. The study further performed has three additional goals: (1) to highlight a performance improvement obtained by a deep-learning based defect predictor when compared to

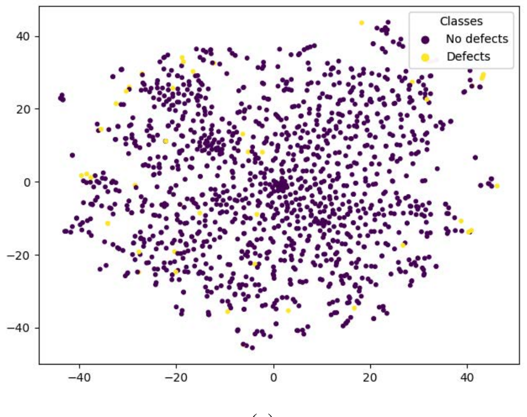


(a)

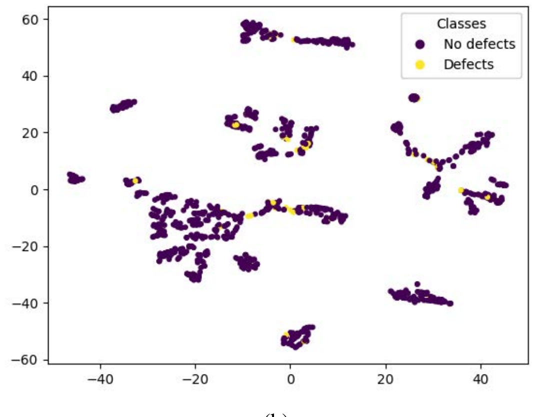


(b)

FIGURE 5. **2D t-SNE visualization for Calcite 1.8.0 for (a) Doc2Vec+LSI and (b) All feature sets.**



(a)



(b)

FIGURE 6. **2D t-SNE visualization for Calcite 1.15.0 for (a) Doc2Vec+LSI and (b) All feature sets.**

classical ML-based defect predictors and thus answering our RQ3; (2) to test the hypothesis that DL-FASTAI brings a statistically significant improvement of the SDP task with respect to the other classifiers; and (3) to highlight the improvement achieved through DL-FASTAI over two baseline classifiers: the *random guessing* and the *Zero rule* baseline.

Apart from the DL-FASTAI model, whose architecture was described in the previous sections, other classifiers have been employed. The ANN, SVM and XGBoost classifiers were selected as the classical ML techniques used as a basis for our comparison as they are well known both in the classical ML literature [85], [86] as well in the SDP literature [8], [87] for their very good predictive performance. One of them is XGBoost, a decision-tree based machine learning algorithm that uses optimised gradient boosting to improve performance [88]. This gradient boosting reduces overfitting by employing regularization and handling of the missing values, as well as parallel processing and tree-pruning. The XGBoost model was also trained using the *FastAI* library [78]. Furthermore, we apply and denote with ANN the scikit-learn implementation of an Artificial Neural Network model. From the same library, we also use a Support Vector Machine (SVM) classifier, a model that constructs a high dimensional hyperplane, in order to find a separation boundary between the two classes [89].

To determine the random guessing baseline, let us denote by d the defective rate (proportion of positive instances) and with n the total number instances in the defect data set (e.g., a given version of the Calcite software). The confusion matrix for the random guessing classifier is the following:

- $TP = n \cdot d^2$, i.e., the number of true positives (defects) for a random guessing classifier is the number of defects $(n \cdot d)$ multiplied with the probability of an instance of being defective (d).
- $TN = n \cdot (1-d)^2$, i.e., the number of true negatives (non-defects) for a random guessing classifier is the number of non-defective entities $n \cdot (1-d)$ multiplied with the probability of an instance of being non-defective $(1-d)$.
- $FP = n \cdot d \cdot (1-d)$, i.e., FP is the number of non-defects $n \cdot (1-d)$ minus the number of true negatives (TN).
- $FN = n \cdot d \cdot (1-d)$, i.e., FN is the number of defects $n \cdot d$ minus the number of true positives (TP).

Based on the previous values, the performance metrics for the random guessing classifier applied on a defect data set with a defective rate of d and the total number of instances n are given in Table 9.

TABLE 9. Performance metrics for the random guessing classifier on a defect data set with a defective rate of *d* and the total number of instances *n*.

Metric	Value	Metric	Value
Acc	$d^2 + (1-d)^2$	$AUPRC$	d
$Prec$	d	MCC	0
NPV	$1-d$	$F\text{-}score^+$	d
$Sens$	d	$F\text{-}score^-$	$1-d$
$Spec$	$1-d$	$F1$	0.5
AUC	0.5	$F1_w$	$d^2 + (1-d)^2$

TABLE 10. Performance metrics for the ZeroR classifier on a defect data set with a defective rate of *d* and the total number of instances *n*.

Metric	Value	Metric	Value
Acc	$1-d$	$AUPRC$	0
$Prec$	0	MCC	0
NPV	$1-d$	$F\text{-}score^+$	0
$Sens$	0	$F\text{-}score^-$	$\frac{2\cdot(1-d)}{2-d}$
$Spec$	1	$F1$	$\frac{1-d}{2-d}$
AUC	0.5	$F1_w$	$\frac{2\cdot(1-d)^2}{2-d}$

The second baseline method we are considering is the Zero rule (ZeroR) classifier. The ZeroR classifier uses the simplest rule of predicting the majority class (i.e., the non-defective class). Considering the same notations previously introduced (d the defective rate and n the total number instances in the defect data set), the confusion matrix for the ZeroR classifier is the following:

- $TP = 0$, since the classifier predicts only the negative class.
- $TN = n \cdot (1-d)$, i.e., all the negative instances (non-defects) are correctly predicted.
- $FP = 0$, since the classifier predicts only the negative class.
- $FN = n \cdot d$, i.e., the number of defects that are misclassified by ZeroR.

The performance metrics for the ZeroR classifier applied on a defect data set with a defective rate of d and the total number of instances n are shown in Table 10.

The performances of the supervised classifiers previously mentioned (DL-FASTAI, XGBoost, SVM, ANN), using the Doc2Vec+LSI feature set and the evaluation metrics described in Section V-A, have been computed for each of the Calcite versions,. Additionally, the evaluation metrics have been determined for the baseline random guessing and ZeroR classifiers, as well. We decided to present the results obtained only for 4 Calcite versions 1.0.0, 1.5.0, 1.8.0 and 1.15.0. Versions 1.0.0, 1.8.0 and 1.15.0 were selected based on the overall difficulty criteria (minimum/median/maximum value), while for version 1.5.0 the best *AUC* has been obtained. The obtained results are given in Table 11. The classifiers have been evaluated using 10-fold cross-validation, the performance metrics being averaged during the 10 runs and 95% CIs being computed for the mean values. Table 11 also includes the performance of the random guessing and ZeroR used as baseline classifiers.

From Table 11 one observes that the best performing model is the DL-FASTAI one. This performance results from the combination of two key factors: a performant ANN based architecture and a state of the art training method provided by the *fastai* library. The other models on our list contain at most one of these factors. Furthermore, it is also worth noting the large improvement in performance of our model over baseline classifiers, whose performances expressed through metrics regarding the positive class (*Precision*, F-*score*$^+$) betray the difficulty of classification on very imbalanced data sets.

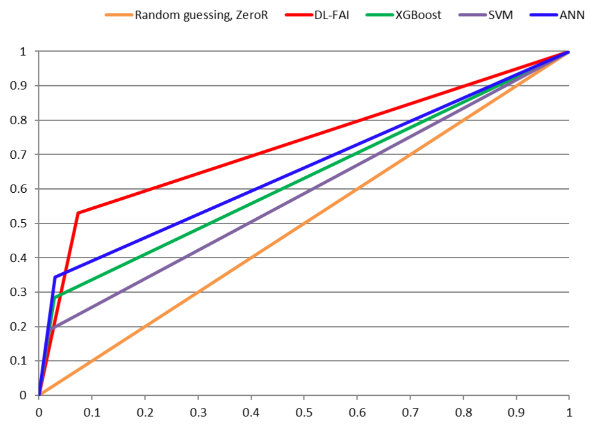


FIGURE 7. ROC curves for the Calcite version 1.5.0.

Table 12 presents, for each classifier $c \in$ {DL-FASTAI, XGBoost, SVM, ANN} (excepting the baselines), the number of Calcite versions for which c provided the best performance considering: (1) all performance metrics (the second column from the table); (2) the sensitivity (*Sens*) metric (the third column from the table); and (3) the *AUC* metric (the last column). We note that for the first evaluation (considering all performance metrics) the best performant classifier c was considered the one that provided the maximum number of performance metrics with the highest value. The second (2) and the third (3) evaluations (considering only the *Sens* and *AUC* measures) have been considered since, as revealed by the SDP literature, a perfomant defect classifier is the one that maximizes *Sens* and *AUC* [79].

The results from Table 12 reveal that the deep learning model DL-FASTAI is the best performing classifier when considering the sensitivity and *AUC* evaluation metrics. Considering all performance metrics, the performance of DL-FASTAI was slightly outperformed by the ANN classifier. Figure 7 presents the ROC curves for the Calcite version 1.5.0 for which the highest *AUC* value has been obtained.

The *AUC* values averaged over all 16 Calcite versions obtained for the evaluated classifiers are given in Table 13. The improvement brought by DL-FASTAI compared to the other classifiers is highlighted in Table 14. In terms of the average *AUC* values, the best performing classifier is DL-FASTAI, which is followed by ANN. In order to answer RQ3, the statistical significance (at a significance level of $\alpha = 0.01$) of the improvement obtained by the DL-FASTAI classifier (in term of AUC values) has been tested against the AUC values provided by XGBoost, SVM and ANN classifiers using a one-tailed paired Wilcoxon signed-rank test. The obtained p-values (0.000241 - DL-FASTAI vs. XGBoost,

TABLE 11. Performance metrics obtained by evaluating DL-FASTAI, XGBoost, SVM and ANN classifiers on calcite versions 1.0.0, 1.5.0, 1.8.0 and 1.15.0 using the Doc2Vec+LSI feature set. 95% CI are used for the results.

Version	Classifier	Acc	$Prec$	NPV	$Sens$	$Spec$	AUC	$AUPRC$	MCC	$F\text{-}score^+$	$F\text{-}score^-$	$F1$	$F1_w$
1.0.0	**Random guessing**	0.723	0.166	0.834	0.166	0.834	0.500	0.166	0.000	0.166	0.834	0.500	0.723
	ZeroR	0.834	0.000	0.834	0.000	1.000	0.500	0.000	0.000	0.000	0.909	0.455	0.759
	DL-FASTAI	0.834 ±0.01	0.473 ± 0.04	0.903 ± 0.02	**0.484 ± 0.08**	0.901 ± 0.01	**0.692 ± 0.04**	0.478 ± 0.05	0.379 ± 0.06	0.472 ± 0.06	0.902 ± 0.01	0.687 ± 0.03	0.834 ± 0.01
	XGBoost	0.844 ±0.01	0.511 ± 0.06	0.893 ± 0.01	0.410 ± 0.04	0.926 ± 0.01	0.668 ± 0.02	0.460 ± 0.05	0.368 ± 0.05	0.452 ± 0.05	0.909 ± 0.01	0.681 ± 0.03	0.837 ± 0.01
	SVM	0.854 ±0.01	0.733 ± 0.07	0.860 ± 0.01	0.206 ± 0.03	0.984 ± 0.00	0.595 ± 0.01	0.469 ± 0.04	0.334 ± 0.04	0.319 ± 0.04	0.918 ± 0.00	0.619 ± 0.02	0.818 ± 0.01
	ANN	0.853 ±0.01	0.582 ± 0.04	0.891 ± 0.01	0.428 ± 0.06	0.939 ± 0.01	0.683 ± 0.03	0.505 ± 0.04	0.415 ± 0.05	0.490 ± 0.05	0.914 ± 0.01	0.702 ± 0.03	0.843 ± 0.01
1.5.0	**Random guessing**	0.839	0.088	0.912	0.088	0.912	0.500	0.088	0.000	0.088	0.912	0.500	0.839
	ZeroR	0.912	0.000	0.912	0.000	1.000	0.500	0.000	0.000	0.000	0.954	0.477	0.870
	DL-FASTAI	0.892 ±0.02	0.393 ± 0.06	0.956 ± 0.01	**0.531 ±0.09**	0.926 ± 0.01	**0.728 ±0.05**	0.462 ±0.07	0.397 ± 0.07	0.447 ± 0.06	0.940 ± 0.01	0.694 ± 0.04	0.899 ± 0.01
	XGBoost	0.914 ±0.01	0.480 ± 0.06	0.937 ± 0.01	0.284 ± 0.05	0.971 ± 0.01	0.628 ± 0.02	0.382 ± 0.05	0.324 ± 0.05	0.352 ± 0.05	0.954 ± 0.01	0.653 ± 0.03	0.904 ± 0.01
	SVM	0.911 ±0.01	0.548 ± 0.14	0.926 ± 0.00	0.190 ± 0.06	0.981 ± 0.01	0.586 ± 0.03	0.369 ± 0.08	0.275 ± 0.07	0.268 ± 0.06	0.953 ± 0.00	0.610 ± 0.03	0.892 ± 0.01
	ANN	0.914 ±0.01	0.535 ± 0.09	0.938 ± 0.01	0.343 ± 0.07	0.970 ± 0.01	0.657 ± 0.03	0.439 ± 0.07	0.383 ± 0.07	0.412 ± 0.07	0.954 ± 0.01	0.683 ± 0.04	0.906 ± 0.01
1.8.0	**Random guessing**	0.856	0.078	0.922	0.078	0.922	0.500	0.078	0.000	0.078	0.922	0.500	0.856
	ZeroR	0.922	0.000	0.922	0.000	1.000	0.500	0.000	0.000	0.000	0.959	0.480	0.885
	DL-FASTAI	0.898 ±0.01	0.397 ± 0.09	0.951 ± 0.01	**0.426 ± 0.11**	0.939 ± 0.02	**0.683 ± 0.05**	0.412 ± 0.06	0.344 ± 0.06	0.378 ± 0.05	0.944 ± 0.01	0.661 ± 0.03	0.901 ± 0.01
	XGBoost	0.918 ±0.01	0.443 ± 0.11	0.936 ± 0.01	0.192 ± 0.04	0.978 ± 0.01	0.585 ± 0.02	0.318 ± 0.07	0.252 ± 0.07	0.264 ± 0.06	0.956 ± 0.01	0.610 ± 0.03	0.904 ± 0.01
	SVM	0.926 ±0.01	0.549 ± 0.09	0.939 ± 0.00	0.225 ± 0.03	0.984 ± 0.00	0.604 ± 0.02	0.387 ± 0.06	0.318 ± 0.06	0.317 ± 0.05	0.961 ± 0.00	0.639 ± 0.02	0.911 ± 0.01
	ANN	0.928 ±0.01	0.564 ± 0.09	0.948 ± 0.00	0.355 ± 0.06	0.976 ± 0.01	0.665 ± 0.03	0.459 ± 0.06	0.408 ± 0.07	0.430 ± 0.06	0.962 ± 0.00	0.696 ± 0.03	0.921 ± 0.01
1.15.0	**Random guessing**	0.936	0.033	0.967	0.033	0.967	0.500	0.033	0.000	0.033	0.967	0.500	0.936
	ZeroR	0.967	0.000	0.967	0.000	1.000	0.500	0.000	0.000	0.000	0.983	0.492	0.951
	DL-FASTAI	0.941 ±0.01	0.274 ± 0.05	0.978 ± 0.01	**0.390 ± 0.11**	0.961 ± 0.01	**0.675 ± 0.05**	0.332 ± 0.06	0.291 ± 0.06	0.308 ± 0.06	0.969 ± 0.01	0.638 ± 0.03	0.947 ± 0.01
	XGBoost	0.966 ±0.00	0.557 ± 0.19	0.972 ± 0.00	0.166 ± 0.05	0.993 ± 0.00	0.580 ± 0.02	0.362 ± 0.10	0.279 ± 0.07	0.242 ± 0.06	0.982 ± 0.00	0.612 ± 0.03	0.958 ± 0.00
	SVM	0.968 ±0.00	0.590 ± 0.13	0.973 ± 0.00	0.200 ± 0.05	0.995 ± 0.00	0.597 ± 0.03	0.395 ± 0.08	0.323 ± 0.07	0.290 ± 0.07	0.984 ± 0.00	0.637 ± 0.04	0.961 ± 0.00
	ANN	0.964 ±0.01	0.498 ± 0.20	0.973 ± 0.00	0.210 ± 0.05	0.989 ± 0.01	0.600 ± 0.03	0.354 ± 0.12	0.299 ± 0.10	0.282 ± 0.08	0.981 ± 0.00	0.632 ± 0.04	0.958 ± 0.01

TABLE 12. The number of calcite versions for which the classifiers provided the best performance considering: (1) all performance metrics; (2) the *Sens* metric; and (3) the *AUC* metric.

Classifier	All metrics	$Sens$	AUC
DL-FASTAI	6	**16**	**14**
XGBoost	0	0	0
SVM	3	0	0
ANN	**7**	0	2

0.000241 for DL-FASTAI vs. SVM, and 0.000512 for DL-FASTAI vs. ANN) reveal a statistically significant improvement acheived by DL-FASTAI, at a significance level of $\alpha = 0.01$.

VII. THREATS TO VALIDITY

In what concerns *construct validity* [90], the performance of the ML models has been analyzed using specific metrics that both stem from literature and characterise the task at hand. However, not for all metrics the models have the same performance ranking and therefore their performance is relative to the task required of them.

When comparing the performance of different models, an *internal validity* pitfall could be focusing only on the architecture and ignoring the different methods used to train those models. Given the fact that the FastAI library contains state of the art training methods, this could lead to erroneous conclusions regarding the convolutional neural network architectures. Furthermore, there is also a comparison of the training methods, by keeping the architecture constant (artificial neural network). In future research, other factors could also be considered when assessing the model performance.

Regarding *external validity* which is concerned with the possibility to generalize the obtained findings, we have chosen a public data set that is relevant for the task of software defect prediction and made public the extracted features. The problem of class imbalancement, present with various degrees in our feature sets, is common in this field area. As further research, our models could be applied to other data sets, to achieve generalization.

TABLE 13. The average of the *AUC* values obtained for all 16 calcite versions and the evaluated classifiers.

DL-FASTAI	Random	ZeroR guessing	XGBoost	SVM	ANN
0.677	0.5	0.5	0.606	0.597	0.640

TABLE 14. The improvement in term of average *AUC* obtained using DL-FASTAI classifier.

Random guessing	ZeroR	XGBoost	SVM	ANN
26.2%	**26.2%**	**10.5%**	**11.9%**	**5.4%**

In order to increase *reliability*, we employed cross validation with 10 repeats of the same experiment, so that statistics would show the most likely result, as well as the confidence interval. The libraries we employed are public and described in the literature, as well as the architectures and training methods. Further analysis could present the value of each parameter used.

VIII. CONCLUSION

In this paper, we conducted an extensive analysis of the impact that different software features have on the performance of software defect predictors. We started from a large set of software features proposed by Herbond *et al.* [10] for SDP and we enlarged it with conceptual software features that are able to capture the semantics of the source code. The conceptual features have been automatically learned using Doc2Vec.

Doc2Vec and LSI models are used only in the feature engineering step, in order to extract conceptual software features that capture the semantics of the source code. These features have been used for enlarging the feature set proposed by Herbold *et al.* [10] in the SDP literature. The enlarged attribute set was then fed into the deep learning model DL-FASTAI (as described in Section V-A) that extracts from the raw input attributes characterizing the software entities a set of features relevant for discriminating between defects and non-defects. The experiments performed in Section VI highlight a statistically significant superior predictive performance of DL-FASTAI with respect to other machine learning models (XGBoost, SVM, ANN). The obtained results empirically validate our hypothesis that the features learned through a deep learning model are better correlated with defect proneness than the raw input attributes and software metrics fed into a classical machine learning model.

A detailed investigation on sixteen different versions of a large scale software system, the Calcite framework, has been performed using both unsupervised and supervised learning-based analyses. The experimental results highlighted a statistically significant improvement obtained on the performance of SDP when using the conceptual features and the deep learning-based predictor we proposed.

The research questions stated in Section I have been answered. First, it has been shown that the performance of software defector prediction can be enhanced by enlarging the classical software features proposed for SDP with conceptual features extracted from the source code. Secondly, the relevance of the conceptual software features for SDP has been highlighted through unsupervised and supervised analyses conducted on Calcite framework. As a third conclusion of our study, a statistically significant improvement has been obtained using a deep-learning based defect predictor instead of traditional supervised classifiers.

For reinforcing the conclusions of the present study we will further investigate other open-source software systems, such as *Apache Commons* libraries (Collections, Compress, Configuration, etc) [91]. We also aim to further extend the feature set for SDP. In this regard we intend to use static analysis tools for source code quality, in order to extract information about software defects, code smells or other source code vulnerabilities. On the other hand, we envision deriving conceptual coupling and cohesion software metrics starting from the proposed conceptual features and investigating their ability to increase the SDP performance even more.

ACKNOWLEDGMENT

The authors would like to thank the editor and the anonymous reviewers for their useful suggestions and comments that helped to improve the article and the presentation.

REFERENCES

[1] R. H. Chang, X. D. Mu, and L. Zhang, "Software defect prediction using non-negative matrix factorization," *J. Softw.*, vol. 6, no. 11, pp. 2114–2120, Nov. 2011.

[2] G. Czibula, Z. Marian, and I. G. Czibula, "Software defect prediction using relational association rule mining," *Inf. Sci.*, vol. 264, pp. 260–278, Apr. 2014.

[3] B. Clark and D. Zubrow, "How good is a software: A review on defect prediction techniques," in *Proc. Softw. Eng. Symp.* Pittsburgh, PA, USA: Carneige Mellon Univ., 2001, pp. 1–35.

[4] J. Zheng, "Predicting software reliability with neural network ensembles," *Exp. Syst. Appl.*, vol. 36, no. 2, pp. 2116–2122, Mar. 2009.

[5] J. Hryszko and L. Madeyski, "Cost effectiveness of software defect prediction in an industrial project," *Found. Comput. Decis. Sci.*, vol. 43, no. 1, pp. 7–35, Mar. 2018. [Online]. Available: https://content.sciendo.com/view/journals/fcds/43/1/article-p7.xml

[6] K. Zhu, N. Zhang, S. Ying, and X. Wang, "Within-project and cross-project software defect prediction based on improved transfer naive Bayes algorithm," *Comput. Mater. Continua*, vol. 63, no. 2, pp. 891–910, 2020.

[7] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic review of fault prediction performance in software engineering," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, Oct. 2011.

[8] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Appl. Soft Comput.*, vol. 27, pp. 504–518, Feb. 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1568494614005857

[9] V. Borisov, T. Leemann, K. Seßler, J. Haug, M. Pawelczyk, and G. Kasneci, "Deep neural networks and tabular data: A survey," 2021, *arXiv:2110.01889*.

[10] S. Herbold, A. Trautsch, F. Trautsch, and B. Ledel, "Problems with SZZ and features: An empirical study of the state of practice of defect prediction data collection," *Empirical Softw. Eng.*, vol. 27, no. 2, pp. 1–49, Jan. 2022 doi: 10.1007/s10664-021-10092-4.

[11] 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.

[12] R. Malhotra, "A defect prediction model for open source software," in *Proc. World Congr. Eng.*, vol. 2, Jul. 2012, pp. 1–5.

[13] W. Afzal, R. Torkar, and R. Feldt, "Resampling methods in software quality classification," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 22, no. 2, pp. 203–223, 2012, doi: 10.1142/S0218194012400037.

[14] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Multi-objective cross-project defect prediction," in *Proc. IEEE 6th Int. Conf. Softw. Test., Verification Validation*, Mar. 2013, pp. 252–261.

[15] X. Xuan, D. Lo, X. Xia, and Y. Tian, "Evaluating defect prediction approaches using a massive set of metrics: An empirical study," in *Proc. 30th Annu. ACM Symp. Appl. Comput.* New York, NY, USA: Association for Computing Machinery, 2015, pp. 1644–1647, doi: 10.1145/2695664.2695959.

[16] Z. Marian, I.-G. Mircea, I.-G. Czibula, and G. Czibula, "A novel approach for software defect prediction using fuzzy decision trees," in *Proc. 18th Int. Symp. Symbolic Numeric Algorithms Sci. Comput. (SYNASC)*, Sep. 2016, pp. 240–247.

[17] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Jul. 2017, pp. 318–328.

[18] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, "A deep tree-based model for software defect prediction," 2018, *arXiv:1802.00921*.

[19] 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.

[20] K. Muthukumaran, S. Srinivas, A. Malapati, and L. B. M. Neti, "Software defect prediction using augmented Bayesian networks," *Adv. Intell. Syst. Comput.*, vol. 614, no. 1, pp. 279–293, 2018.

[21] D.-L. Miholca, G. Czibula, and I. G. Czibula, "A novel approach for software defect prediction through hybridizing gradual relational association rules with artificial neural networks," *Inf. Sci.*, vol. 441, pp. 152–170, May 2018, doi: 10.1016/j.ins.2018.02.027.

[22] D.-L. Miholca, "An improved approach to software defect prediction using a hybrid machine learning model," in *Proc. 20th Int. Symp. Symbolic Numeric Algorithms Sci. Comput. (SYNASC)*, Sep. 2018, pp. 443–448.

[23] S. Sayyad and T. Menzies, "The PROMISE repository of software engineering databases," School Inf. Technol. Eng., Univ. Ottawa, Ottawa, ON, Canada, Tech. Rep. 2015. [Online]. Available: http://promise.site.uottawa.ca/SERepository

[24] "The seacraft repository of empirical software engineering data," Tech. Rep., 2017.

[25] S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object oriented design," *ACM SIGPLAN Notices*, vol. 26, no. 11, pp. 197–211, Nov. 1991, doi: 10.1145/118014.117970.

[26] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[27] *The Metrics Data Program Data Repository*.

[28] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proc. 3rd Int. Workshop Predictor Models Softw. Eng. (PROMISE, ICSE Workshops)*, May 2007, p. 1–7.

[29] 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, 2009, doi: 10.1007/s10664-008-9103-7.

[30] 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.* New York, NY, USA: Association for Computing Machinery, 2011, pp. 15–25, doi: 10.1145/2025113.2025120.

[31] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empirical Softw. Engg.*, vol. 17, nos. 4–5, pp. 531–577, Aug. 2012, doi: 10.1007/s10664-011-9173-9.

[32] K. Herzig, S. Just, A. Rau, and A. Zeller, "Predicting defects using change genealogies," in *Proc. IEEE 24th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Nov. 2013, pp. 118–127.

[33] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, Jun. 2013.

[34] L. Madeyski and M. Jureczko, "Which process metrics can significantly improve defect prediction models? An empirical study," *Softw. Qual. J.*, vol. 23, no. 3, pp. 393–422, Sep. 2015, doi: 10.1007/s11219-014-9241-7.

[35] H. Altinger, S. Siegl, Y. Dajsuren, and F. Wotawa, "A novel industry grade dataset for fault prediction based on model-driven developed automotive embedded software," in *Proc. IEEE/ACM 12th Work. Conf. Mining Softw. Repositories*, May 2015, pp. 494–497.

[36] T. Shippey, T. Hall, S. Counsell, and D. Bowes, "So you need more method level datasets for your software defect prediction? Voilá!" in *Proc. 10th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.* New York, NY, USA: Association for Computing Machinery, Sep. 2016, pp. 1–6, doi: 10.1145/2961111.2962620.

[37] Z. Toth, P. Gyimesi, and R. Ferenc, "A public bug database of Github projects and its application in bug prediction," in *Proc. Int. Conf. Comput. Sci. Appl.*, 2016, vol. 9789, no. 7, pp. 625–638.

[38] S. Yatish, J. Jiarpakdee, P. Thongtanunam, and C. Tantithamthavorn, "Mining software defects: Should we consider affected releases?" in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 654–665.

[39] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *J. Syst. Softw.*, vol. 150, pp. 22–36, Apr. 2019.

[40] R. Ferenc, P. Gyimesi, G. Gyimesi, Z. Tóth, and T. Gyimóthy, "An automatically created novel bug dataset and its validation in bug prediction," *J. Syst. Softw.*, vol. 169, Nov. 2020, Art. no. 110691, doi: 10.1016/j.jss.2020.110691.

[41] R. B. Grady, *Practical Software Metrics for Project Management and Process Improvement*. Upper Saddle River, NJ, USA: Prentice-Hall, 1992.

[42] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. Int. Workshop Mining Softw. Repositories*. New York, NY, USA: Association for Computing Machinery, 2005, pp. 1–5, doi: 10.1145/1083142.1083147.

[43] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proc. 3rd Int. Workshop Predictor Models Softw. Eng. (PROMISE, ICSE Workshops)*, May 2007, p. 1–11.

[44] C. Mills, J. Pantiuchina, E. Parra, G. Bavota, and S. Haiduc, "Are bug reports enough for text retrieval-based bug localization?" in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2018, pp. 381–392.

[45] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes," *IEEE Trans. Softw. Eng.*, vol. 43, no. 7, pp. 641–657, Jul. 2017.

[46] E. C. Neto, D. A. da Costa, and U. Kulesza, "The impact of refactoring changes on the SZZ algorithm: An empirical study," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Mar. 2018, pp. 380–390.

[47] R. Malhotra, "Comparative analysis of statistical and machine learning methods for predicting faulty modules," *Appl. Soft Comput.*, vol. 21, pp. 286–297, Aug. 2014.

[48] A. Panichella, R. Oliveto, and A. De Lucia, "Cross-project defect prediction models: L'union fait La force," in *Proc. Softw. Evol. Week IEEE Conf. Softw. Maintenance, Reeng., Reverse Eng. (CSMR-WCRE)*, Feb. 2014, pp. 164–173.

[49] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: An update mark," *ACM SIGKDD Explor. Newslett.*, vol. 11, no. 1, pp. 10–18, 2009.

[50] L. Yu and A. Mishra, "Experience in predicting fault-prone software modules using complexity metrics," *Qual. Technol. Quant. Manage.*, vol. 9, no. 4, pp. 421–433, 2012.

[51] J. Nam and S. Kim, "Heterogeneous defect prediction," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 508–519, doi: 10.1145/2786805.2786814.

[52] S. Tiwari and S. S. Rathore, "Coupling and cohesion metrics for object-oriented software: A systematic mapping study," in *Proc. 11th Innov. Softw. Eng. Conf.* New York, NY, USA: Association for Computing Machinery, 2018, pp. 1–11.

[53] D.-L. Miholca, G. Czibula, and V. Tomescu, "COMET: A conceptual coupling based metrics suite for software defect prediction," *Proc. Comput. Sci.*, vol. 176, pp. 31–40, Jan. 2020.

[54] D.-L. Miholca and Z. Onet-Marian, "An analysis of aggregated coupling's suitability for software defect prediction," in *Proc. 22nd Int. Symp. Symbolic Numeric Algorithms Sci. Comput. (SYNASC)*, Sep. 2020, pp. 141–148.

[55] S. S. Rathore and S. Kumar, "A study on software fault prediction techniques," *Artif. Intell. Rev.*, vol. 51, no. 2, pp. 255–327, Feb. 2019.

[56] D.-L. Miholca and G. Czibula, "Software defect prediction using a hybrid model based on semantic features learned from the source code," in *Proc. Int. Conf. Knowl. Sci., Eng. Manage.* Berlin, Germany: Springer, Aug. 2019, pp. 262–274.

[57] 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.

[58] L. Šikić, A. S. Kurdija, K. Vladimir, and M. Šikić, "Graph neural network for source code defect prediction," *IEEE Access*, vol. 10, pp. 10402–10415, 2022.

[59] L. Zhao, Z. Shang, A. Qin, and Y. Tang, "Siamese dense neural network for software defect prediction with small data," *IEEE Access*, vol. 7, pp. 7663–7677, 2018.

[60] L. Zhao, Z. Shang, L. Zhao, T. Zhang, and Y. Tang, "Software defect prediction via cost-sensitive Siamese parallel fully-connected neural networks," *Neurocomputing*, vol. 352, pp. 64–74, Aug. 2019.

[61] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire, "Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources," in *Proc. Int. Conf. Manage. Data (SIGMOD)*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 221–230, doi: 10.1145/3183713.3190662.

[62] V. I. Tomescu, "Calcite data set with Boolean label," Tech. Rep., 2022, doi: 10.7910/DVN/VDBQVV.

[63] *Shark*. [Online]. Available: https://github.com/smartshark

[64] *PMD—An Extensible Cross-Language Static Code Analyzer*. [Online]. Available: https://pmd.github.io/

[65] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 725–743, Nov. 2007.
[66] D. Silva and M. T. Valente, "RefDiff: Detecting refactorings in version histories," in *Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories (MSR)*, May 2017, pp. 269–279.
[67] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proc. 30th Int. Conf. Softw. Eng.* New York, NY, USA: Association for Computing Machinery, 2008, pp. 181–190, doi: 10.1145/1368088.1368114.
[68] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, May 2009, pp. 78–88.
[69] F. Zhang, A. E. Hassan, S. McIntosh, and Y. Zou, "The use of summation to aggregate software metrics hinders the performance of defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 5, pp. 476–491, May 2017.
[70] D. Zhang, J. Tsai, and G. Boetticher, "Improving credibility of machine learner models in software engineering," in *Proc. Adv. Mach. Learn. Appl. Softw. Eng.*, 2007, pp. 52–72.
[71] Y. Dodge, *The Concise Encyclopedia of Statistics*. Cham, Switzerland: Springer, 2008.
[72] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *Empirical Softw. Eng.*, vol. 14, no. 1, pp. 5–32, Feb. 2009.
[73] I. G. Czibula, G. Czibula, D.-L. Miholca, and Z. Onet-Marian, "An aggregated coupling measure for the analysis of object-oriented software systems," *J. Syst. Softw.*, vol. 148, pp. 1–20, Feb. 2019.
[74] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur.*, Aug. 2015, pp. 17–26.
[75] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," 2014, *arXiv:1405.4053*.
[76] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *J. Amer. Soc. Inf. Sci.*, vol. 41, no. 6, pp. 391–407, 1990.
[77] R. Řehůřek and P. Sojka, "Software framework for topic modelling with large corpora," in *Proc. LREC Workshop New Challenges NLP Frameworks*, 2010, pp. 45–50.
[78] J. Howard *et al.* (2018). *Fastai*. [Online]. Available: https://github.com/fastai/fastai
[79] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognit. Lett.*, vol. 27, no. 8, pp. 861–874, Jun. 2006.
[80] S. Boughorbel, F. Jarray, and M. El-Anbari, "Optimal classifier for imbalanced data using Matthews correlation coefficient metric," *PLoS ONE*, vol. 12, no. 6, 2017, Art. no. e0177678.
[81] "Inferential statistics III: Nonparametric hypothesis testing," in *Statistics for Biomedical Engineers and Scientists*, A. P. King and R. J. Eckersley, Eds. New York, NY, USA: Academic, 2019, ch. 6, pp. 119–145.
[82] *Social Science Statistics*. [Online]. Available: http://www.socscistatistics.com/tests/
[83] L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," *J. Mach. Learn. Res.*, vol. 9, pp. 2579–2605, Nov. 2008.
[84] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, A. Müller, J. Nothman, G. Louppe, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Oct. 2011.
[85] T. Eitrich and B. Lang, "Efficient optimization of support vector machine learning parameters for unbalanced datasets," *J. Comput. Appl. Math.*, vol. 196, no. 2, pp. 425–436, 2006.
[86] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*. New York, NY, USA: Association for Computing Machinery, Aug. 2016, pp. 785–794.
[87] G. Santos, E. Figueiredo, A. Veloso, M. Viggiato, and N. Ziviani, "Predicting software defects with explainable machine learning," in *Proc. 19th Brazilian Symp. Softw. Qual.* New York, NY, USA: Association for Computing Machinery, Dec. 2020, pp. 1–10.
[88] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2016, pp. 785–794, doi: 10.1145/2939672.2939785.
[89] D. A. Pisner and D. M. Schnyer, "Support vector machine," in *Machine Learning*, A. Mechelli and S. Vieira, Eds. New York, NY, USA: Academic, 2020, ch. 6, pp. 101–121.
[90] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Eng.*, vol. 14, no. 2, pp. 131–164, Apr. 2009, doi: 10.1007/s10664-008-9102-8.
[91] *Apache Commons*. [Online]. Available: https://commons.apache.org/

DIANA-LUCIA MIHOLCA received the Ph.D. degree in computer science, in 2020. She is an Assistant with the Department of Computer Science, Faculty of Mathematics and Computer Science, Babes-Bolyai University, Cluj-Napoca, Romania. She has published 17 papers in international journals and conference proceedings. Her research interests include computational intelligence, machine learning, search-based software engineering, and data mining.

VLAD-IOAN TOMESCU is currently pursuing the Ph.D. degree with the Faculty of Mathematics and Computer Science, Babes-Bolyai University, Cluj-Napoca, Romania. He has published five papers in conference proceedings and journals. His main research interests include machine learning, computer vision, and search-based software engineering.

GABRIELA CZIBULA is a Professor with the Computer Science Department, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania. She has published more than 200 papers in prestigious journals and conferences proceedings. Her research interests include machine learning, distributed artificial intelligence, multiagent systems, and bioinformatics.