IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 30, 2021, accepted January 15, 2022, date of publication January 18, 2022, date of current version January 28, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3144598

Graph Neural Network for Source Code Defect Prediction

LUCIJA ŠIKIĆ, ADRIAN SATJA KURDIJA, (Member, IEEE), KLEMO VLADIMIR, (Member, IEEE), AND MARIN ŠILIĆ, (Member, IEEE)
Faculty of Electrical Engineering and Computing, University of Zagreb, 10000 Zagreb, Croatia

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

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

ABSTRACT Predicting defective software modules before testing is a useful operation that ensures that the time and cost of software testing can be reduced. In recent years, several models have been proposed for this purpose, most of which are built using deep learning-based methods. However, most of these models do not take full advantage of a source code as they ignore its tree structure or they focus only on a small part of a code. To investigate whether and to what extent information from this structure can be beneficial in predicting defective source code, we developed an end-to-end model based on a convolutional graph neural network (GCNN) for defect prediction, whose architecture can be adapted to the analyzed software, so that projects of different sizes can be processed with the same level of detail. The model processes the information of the nodes and edges from the abstract syntax tree (AST) of the source code of a software module and classifies the module as defective or not defective based on this information. Experiments on open source projects written in Java have shown that the proposed model performs significantly better than traditional defect prediction models in terms of AUC and F-score. Based on the F-scores of the existing *state-of-the-art* models, the model has shown comparable predictive capabilities for the analyzed projects.

INDEX TERMS Software defect prediction, deep learning, graph neural network.

I. INTRODUCTION

The development of source code defect prediction models plays an important role in improving software quality. Such models are used to identify defective software modules so that they can be corrected before the testing process, which ultimately helps to optimize the allocation of testing resources.

Software defect prediction (SDP) models classify software modules based on the features used to represent them. Traditionally, the features are manually designed from the qualitative or quantitative description of the module or its development process. However, these features ignore both the unambiguous syntax and semantics that define a programming language used for software development and that provide additional information about the software modules. This reason, combined with the suspicion that models using traditional features have reached a performance limit [1], has led recent research to focus on developing a model that extracts features from the source code of software modules. These models are developed using deep learning techniques, which enables them to automatically learn complicated hidden patterns from high-dimensional data [2] such as the Abstract Syntax Trees (ASTs). In recent years, several deep learning-based models have been proposed to extract features from ASTs to identify defective source code. So far, the Deep Belief Network- [3], Convolutional Neural Network- [4]–[7], Long-Short Term Memory Network- [8], [9], Recurrent Neural Network [10], Encoder-Decoder-based model [11], and GCNNs [12], [13] have been proposed for this purpose.

Despite the fact that the proposed defect prediction models extract features to represent software modules from ASTs of the modules' source code, the vast majority of them treat ASTs as linear sequences of nodes and use natural language processing models to generate embedding vectors of these sequences. In this way, they ignore the structure of ASTs and thus miss the opportunity to use additional

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

information about source code that could be used to generate more expressive features than when the features are generated from the linear sequences of nodes from AST. Indeed, recent research has shown that extracting features from a representation of source code in graph or tree form rather than in a sequence of AST's nodes gives better results in various tasks, such as code classification [14], [15], code clone detection [14], [16], [17], plagiarism detection [18], [19], programmers' de-anonymization [20], [21], variable naming [22], etc. Although AST-based methods are used for the tasks, none of these methods are specialized for addressing the task of predicting defective software considered in this work.

Although there are some approaches for predicting defective software modules that encode the structural information of ASTs into features for describing modules, shortcomings can be observed in these approaches. In particular, such approaches compute features either on a bottom-up basis [8], using specially designed relationships between AST's nodes [6], or focuses only on a part of an AST [12]. The bottom-up method used by Dam *et al.* [8] computes the features of AST from leaf nodes to root nodes, which makes it difficult to capture the long-range dependencies between distant nodes. Furthermore, it is not entirely clear why it is necessary to define additional relationships in a tree besides edges, as done by Shi *et al.* [6], since such relationships do not provide any new information about the tree itself or about the defectiveness of the source code. Moreover, using only a part of an AST to represent the whole software module, as in work by Xu *et al.* [12], may be memory efficient, but it is indisputable that a valuable information about software module can be lost when neglecting the whole structure of the AST representing its source code. Finally, Zhao *et al.* [13] recently proved that the features extracted from GNN can be useful for traditional classifiers. They combined features from the Control Flow Graph (CFG) and AST-based local features obtained from TBCNN with contextual feature vectors extracted from an Attention-based Graph Neural Network for Directed Graph to represent a source code by a feature vector. However, they limit the size of GNN layers and vectors so that the representation of source codes of different sizes has the same size, which may lead to a loss of information.

To address this gap, we developed an end-to-end defect prediction model based on the Convolutional Graph Neural Network (GCNN), a class of neural network models capable of effectively categorizing graphs by learning their representations. Unlike the existing models that predict defective software modules based on the ASTs from the modules' source code, our model uses a neural network architecture that is specifically tailored for graph data, which includes ASTs, and for task of defect prediction. The proposed model captures the information relevant to source code defectiveness from data representing ASTs of the source code, and the information is then used by the classification layer that is on top of the model architecture. By training the model in a supervised manner, we make it well-suited to the task it is being used for, which is an opportunity missed when feature extraction and classification are performed separately, as in many existing studies [3]–[8], [11], [12].

The developed model learns to identify defective software modules by relying on the GCNN's ability to use relatively unstructured data types, such as ASTs, as input data. In particular, by using the graph representation of ASTs as input data, the full information of the modules' source code can be preserved, which is not necessarily the case when ASTs must first be transformed into the form suitable for the feature extraction and/or classification model. In other words, by using a model suitable for the data, we can bypass a transformation step and process the data in its most natural form without risking the loss of valuable information.

More specifically, the model uses the graph representations of ASTs created from the source code of the software modules. The representations are generated by translating ASTs into a pair of two matrices: the adjacency matrix and the feature matrix. The elements of the adjacency matrix indicate whether the pairs of nodes of AST are adjacent or not, while the feature matrix describes the nodes of AST. These matrices are forwarded as an input for the spectral-based GCNN which is trained to identify defective software modules of the future project version. The performance of the model in terms of F-score and AUC score is evaluated on seven open source Java projects commonly used in SDP studies and compared with the performance of traditional SDP models. The comparison shows that the model performs best in both aspects. In addition, in terms of F-score, the model gives comparable results to *state-of-the-art* AST-based approaches.

In summary, this work makes the following contributions:

- We propose an end-to-end SDP model that identifies defective software modules by using GCNN to capture the entire information of ASTs representing the source code of the modules.
- In the experiments conducted, we have shown that the proposed model outperforms the traditional SDP models and provides comparable results to the *state-of-the-art* models for predicting defective software modules using ASTs from the source code of the modules.

The rest of the paper is organized as follows. Section II introduces the background of SDP and GCNNs. Section III describes the proposed model. Section IV describes the design of the experiment conducted and the results obtained, with a comparison to relevant work. Section V discusses related work. In Section VI the threats to validity are listed. Conclusion and future work are presented in Section VII.

II. BACKGROUND

In Section II-A, we present the basics of SDP. GCNNs and the motivation of using such networks in software defect prediction are briefly described in Section II-B.

A. SOFTWARE DEFECT PREDICTION

SDP is an important area in software engineering research. Great efforts in this area have been devoted to the

development of models for estimating the number of defects, discovering defective patterns, and classifying the defect-proneness of software modules, typically into defect-prone and non defect-prone [23]. This study focuses on the latter of these tasks, which is therefore referred to as ''defect prediction'' hereafter.

The task of defect prediction typically consists of three subtasks: creating or/and collecting features, selecting a subset of the features which is the most relevant to the defect label, and choosing a suitable classifier. The first subtask is taken before the other two steps. The second subtask can be performed before or within the third subtask, which depends on how the decision on the features to be used is made.

Depending on whether the aim is to identify defect-prone software modules or changes, two types of defect prediction can be distinguished: file-level and just-in-time [24]. File-level defect prediction is usually performed before a product release, while just-in-time defect prediction is conducted after each file change. As they are used in different development phases, the corresponding features are also different. Besides, defect prediction models can be categorized based on their purpose. Specifically, if they are used to detect defect-prone software modules within the project whose historical data is used for their development, they belong to the Within-Project Defect Prediction (WPDP) models and to the Cross-Project Data Prediction (CPDP) otherwise. The WPDP models are mainly trained on the previous project version and then used to predict defective modules in the current version, which is called cross-version defect prediction. These particular models have achieved remarkable results in file-level defect prediction [3], [4], [8], [10], [17], [25], [26] we are dealing with in this study.

For file-level defect prediction, two types of features can be distinguished: process features, which describes the software development process, and code features, which reflect code properties. Process features can be divided into developer metrics, code change metrics, and development process metrics [27]. Code features include static code metrics, object-oriented metrics, network metrics, and source code features, which are extracted from a source code represented by an AST or a control flow graph (CFG).

Together with corresponding defect labels, features that represent software modules are generally used as an input to a classifier, which is trained to identify defect-prone modules. The classifier is typically chosen depending on the type and size of the features representing modules, whereby the features can be selected independently from the classifier or as a part of its development.

B. CONVOLUTIONAL GRAPH NEURAL NETWORKS

GCNNs are a type of graph neural networks that adapt the operation of convolution from grid data to graph data. In contrast to convolutional neural networks (CNNs), they do not have strict structural requirements for graphs that are fed as input, which makes them suitable for handling unstructured data [28].

Depending on how the convolutional operation is adjusted, two categories of GCNNs can be distinguished: spectral-based and spatial-based. Spectral-based GCNNs operate on the principle from graph signal processing where a graph is transformed into the spectral domain within which it can be convolved with a filter [29]. On the other hand, spatial-based GCNNs define a convolution operation based on a node's spatial relations in the graph domain. In this context, applying the convolution filter on a feature vector representing a graph node may be perceived as selecting and aggregating information of nodes that are in a predefined neighborhood of the node.

An input of a GCNN-based model consists of the adjacency matrix A[1] and the node feature matrix $X_v, v \in V$ of graph G with a set of vertices $V = \{v_1, v_2, \ldots, v_n\}$. The adjacency matrix is used by spatial-based GCNNs to calculate nodes' neighborhoods, while spectral-based GCNNs use it to transform the graph into the spectral domain. Specifically, spectral-based GCNNs perform convolution using the eigenvalue decomposition of the symmetrically normalized Laplacian matrix $\tilde{L} = I - D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ of A, where I is an identity matrix and a diagonal matrix D is the degree matrix of A with $D(i, i) = \sum_{j=1}^{n} A(i, j)$.

The output and therefore also the architecture of both types of GCNN-based models depend on whether they are developed to perform node- or graph-level classification. During the training for node classification, a model is adjusting its parameters' values that are used to generate representations of nodes such that the nodes' labels can be predicted from the generated representations. Precisely, the representation of node v from the $(l+1)$-th layer $h_v^{(l+1)}$ of a spatial-based GCNN can be expressed with Eq. (1), where $h^{(l)}$ denotes representations of nodes from the l-th layer, $\mathcal{N}(v)$ is a set of neighbors of node v, V and W are the model's parameters to be learned, $\psi(\cdot)$ is a permutation-equivariant function, $g(\cdot)$ is a permutation-invariant aggregation function, and $\phi(\cdot)$ is any function that can be applied on the representation values.

$$h_v^{(l+1)} = \phi\left(W^{(l)},\ h_v^{(l)},\ g\left(\left\{V_{vu}^{(l)}\psi\left(h_u^{(l)}\right) \mid u \in \mathcal{N}(v)\right\}\right)\right) \tag{1}$$

Similarly, using the same notation as in Eq. (1), the representation of node v from the $(l+1)$-th layer of a spectral-based GCNN can be calculated using Eq. (2), where $W^{(l)}(\cdot)$ is a function whose parameters are to be learned.

$$h_v^{(l+1)} = \phi\left(W_v^{(l)}(\tilde{L}),\ h^{(l)}\right) \tag{2}$$

A model trained for graph classification aims to learn the representation of a graph that helps predict the label of the entire graph. A convolutional layer of such model is usually followed by a pooling layer. As a result, graph is coarsened

[1]The adjacency matrix A is defined as follows: if there is an edge between node u and v, then $A(u, v)$ is equal to the weight of the edge (or 1 if graph G is unweighted); otherwise it is equal to zero.

through layers so, effectively, a convolutional layer encapsulates sub-graph representations. These representations are summarized into the final graph representation in a readout layer by aggregating representations of sub-graphs. Formally, if L is the final graph convolutional layer of the GCNN-based model's architecture, then the entire graph's representation h_G can be obtained using Eq. (3), where $\mathcal{S}_{L-1}$ denotes a set of representations of sub-graphs from the $(L-1)$-th layer and U is the model's parameter to be learned.

$$h_G = g\left(\left\{U_s^{(L)}\psi\left(h_s^{(L)}\right) \mid s \in \mathcal{S}_{L-1}\right\}\right) \tag{3}$$

The entire graph's representation h_G is given as an input to a stack of fully connected layers of the GCNN-based model, which produce the classification output.

III. PROPOSED MODEL

In this section, we describe preparation of data for the proposed model and the model itself. In subsection III-A, we describe how ASTs are used to represent a source code of a software module. The ASTs are used as an input to the proposed model for predicting defective software modules, which is described in subsection III-B.

A. REPRESENTING SOFTWARE MODULES WITH ASTs

An AST is a model of source code which represents a program on the level of abstract syntax, meaning it does not capture all the details appearing in the code, but rather only structural and content-related details. Specifically, each node of an AST denotes a construct, such as `MethodDeclaration`, `IfStatement`, `VariableAccess`, etc., occurring in the source code, as shown in Figure 1.

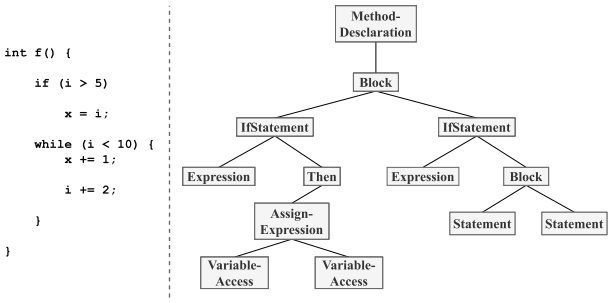


FIGURE 1. An example of a Java method and its abstract syntax tree.

To obtain an AST from the source code of the analyzed software module, we employ an open-source Python library javalang,[2] which provides a lexer and a parser for Java programming language. There are numerous types of AST nodes within the library, but almost all recent studies, except for study by Xu *et al.* [12], have excluded some nodes from the analysis because they do not provide information that may indicate a source code being defective, such as `PackageDeclaration` and `Type` nodes. Similarly, we have chosen 39 AST nodes defined in javalang that can be categorized as shown in Table 1. However, as due to the process of selecting only certain nodes an AST can fragment into two or more sub-graphs,[3] we added a `root` node and an edge between `root` and a root node of each isolated sub-graph, thus ensuring each AST is being connected.

TABLE 1. AST nodes considered in this work.

`Class-`, `Enum-`, `EnumConstant-`, `Interface-`, `Annotation-`, `Method-`, `Field-`, `Constructor-`, `Constant-`, `LocalVariable-`, `Variable-Declaration`, `VariableDeclarator`, `If-`, `While-`, `Do-`, `For-`, `Assert-`, `Break-`, `Continue-`, `Return-`, `Throw-`, `Synchronized-`, `Try-`, `Switch-`, `Block-Statement`, `ExplicitConstructor-`, `SuperConstructor-`, `Method-`, `SuperMethod-Invocation`, `Method-`, `Member-`, `SuperMember-`, `Class-`, `VoidClass-Reference`, `Array-`, `Class-`, `InnerClass-Creator`, `Annotation`, `ArrayInitializer`

The proposed GCNN-based model works with numerical data, so it is necessary to encode the AST nodes represented with string tokens, into a suitable data structure. Using the AST nodes listed in Table 1 and an additional node representing the `root` node, we decided to use one-hot encoding, which is defined in accordance with the order of the AST nodes listed in Table 1. In this setting, each of the AST nodes is represented by a one-hot vector of size 40.

B. GCNN-BASED MODEL

Along with the corresponding defect labels, ASTs are used to train a spectral-based GCNN to identify defect-prone software modules. The architecture of the proposed GCNN is adaptive such so the size of its layers depends on the maximum number of nodes in the ASTs from the project being analyzed. Accordingly, the padding nodes, which are represented with all-zeros feature vector, are added to all ASTs from the project so that they have a size equal to the number of nodes in the largest AST. Fitting the architecture to the data allows the model to exploit the data better than if the data were to be compressed or expanded into a form predefined by the model.

Processing an AST using the spectral-based GCNN requires the AST to be represented with the corresponding adjacency matrix A and feature matrix X. The adjacency matrix A is used to calculate the symmetrically normalized Laplacian matrix $\tilde{L}$, which is further used to create a filter for convolutional layers. To reduce the learning complexity of the model, we followed the approach proposed by Defferrard *et al.* [30] and approximated the spectral convolutional filter with the Chebyshev polynomial $T_k(x)$ of order k, where $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$ with $T_0 = 1$ and $T_1 = x$. Accordingly, the convolutional operation of the

[2]https://github.com/c2nes/javalang

[3]It may happen, for instance, when there is an annotation in a source code.

l-th layer from the Eq. 2 with the filter function $W^{(l)}(\tilde{L})$ is approximated with $\sum_{k=0}^{K-1} \Theta_k T_k(\tilde{L})$, where $\Theta_k \in \mathbf{R}^K$ is a vector of polynomial coefficients to be learned.

The proposed model's architecture is shown in the Fig. 2. Between the input layer and the output layer of the model there is a sufficient number of levels of alternating layers of convolution and pooling, whereby each level is followed by an dropout layer. The model generates hierarchical graph representations within these levels in a way that it first generates features of an input graph's nodes in its convolutional layer and then reduces the size of the graph in the following pooling layer. After the last pooling layer, there is a pooling layer that computes the average node features of the highest level representation of the input AST. Such features are fully connected to the last hidden layer, which is followed by the output layer with sigmoid non-linearity to predict the probability of the input AST representing defective source code.

We have designed the model to be able to capture the same amount of information about each AST. To this aim, sizes of the model's layers are defined to be dependent on the largest number of nodes in an AST from the project. Specifically, in a convolutional layer, the number of features representing each node is equal to the size of a graph which inputs the layer, while a pooling layer halves the number of nodes in the graph. The number of neurons in the last hidden layer is set to a half of the number of nodes from the AST that is the output of the last pooling layer.

To retain the possibility of training the proposed model end-to-end, its pooling layers should be differentiable. Therefore, we decided to use the MinCutPool pooling layer [31], whose parameters are learned by minimizing the relaxed formulation of the normalized mincut problem.[4]

IV. EXPERIMENTS AND RESULTS

In this section, we describe experiments conducted to evaluate the performance of the proposed defect prediction model. The experimental results should allow us to determine if the proposed model outperforms the *state-of-the-art* models that are based on information from ASTs generated from the modules' source code in cross-version defect prediction.

The experimental results should answer the following research questions (RQ):

RQ1: Does the proposed model improve the performance of the commonly used classifiers based on the traditional code features in cross-version defect prediction?

RQ2: Is the performance of the proposed model comparable to the *state-of-the-art* models in cross-version defect prediction in terms of F-score?

The proposed model has been implemented using Keras, a Python library for deep learning, on top of Tensorflow back end. It was trained using the Adam [32] algorithm and binary cross-entropy loss function for 200 epochs. In order to avoid overfitting of the model, we randomly selected 10% of training set as a validation set and used the early stopping technique [33] with a patience of 20 so the training stops if the loss on the validation set does not decrease for 20 consecutive epochs. Other training parameters are to be discussed in Section IV-C. The commonly used classifiers were created using scikit-learn modules. All experiments were run on a NVIDIA GeForce Titan Xp GPU with RAM of 32 GB. The implementation of the proposed model and the experimental results are available at GitLab.[5]

In the following subsections, details of the experimental setup and results are given. Section IV-A describes the data set used within experiments. Section IV-B defines measures used to evaluate models' performances. Section IV-C describes how the models' parameters are set. In Section IV-D baseline methods are listed. In Section IV-E the obtained results are shown and discussed.

A. DATA SET

The data set used in this work consists of source code from seven Java Apache projects collected from PROMISE,[6] a publicly accessible repository of SDP research data collected by Jureczko and Madeyski [34]. Specifically, each project version from PROMISE is represented by a list of classes it consists of, and each class is described by 20 traditional code features, such as lines of code, and the defect label. The PROMISE repository does not contain the classes' source codes that are required to generate ASTs, so we downloaded the corresponding versions of the projects from open source repositories and extracted the source code from the open-source project files.

As with the most relevant approaches [3], [4], [8], [10], [17], [25], [26], we chose two consecutive versions of seven projects from PROMISE for the purpose of the experiments. All files from the earlier version of the project were used to create the ASTs used for a training data set, while the ASTs from all files in the latest version were used as a test data set. Detailed information about projects and versions used in this work can be found in Table 2.

B. EVALUATION MEASURES

The performance of the SDP model is typically evaluated using Precision, Recall, F-score and Area Under the ROC Curve (AUC). To be able to compare with the *state-of-the-art* models' performances, we evaluated our model using F-score. However, we have reported AUC score as well because it has a lower variance, i.e. it is more static than any of the above metrics, and is therefore highly preferable for the evaluation of defect prediction models [35].

The F-score is the harmonic mean of precision and recall. It is a widely used measure of the accuracy of the test, with values between 0 for the worst accuracy and 1 for the best accuracy.

[4]The K-way normalized mincut problem is the task of partitioning graph vertices in K disjoint subsets by removing the minimum volume of edges.

[5]https://gitlab.com/LSikic/dp-gcnn

[6]https://github.com/opensciences/opensciences.github.io

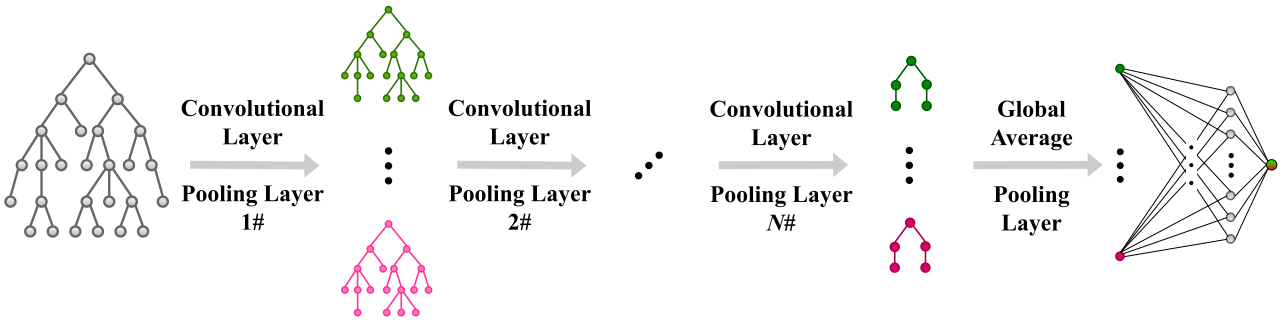


FIGURE 2. The proposed defect prediction model's architecture.

TABLE 2. Statistical data of projects from PROMISE used in the experiments.

Project	Version	#Files	Defects [%]	#Nodes
camel	1.4	872	16.63	77
	1.6	965	19.48	82
jedit	4.0	306	22.60	248
	4.1	312	25.32	260
lucene	2.0	195	46.67	167
	2.2	247	58.30	162
poi	2.5	385	64.42	183
	3.0	442	63.57	174
synapse	1.1	222	27.03	125
	1.2	256	33.59	135
xalan	2.5	803	48.19	202
	2.6	885	46.44	229
xerces	1.2	440	16.14	218
	1.3	453	15.23	226

The AUC is based on the area under the ROC (Receiver Operating Characteristic). A model whose predictions are completely wrong has an AUC of 0, while a model whose predictions are completely correct has an AUC of 1. It is therefore said that the AUC score measures how well the developed model can distinguish between the defect class and the non-defect class.

C. HYPERPARAMETER SETTING

Before conducting a study, the parameters of the machine learning model should be tuned, as this is essential for optimizing its performance [36]. Therefore, we tuned the following parameters of the model: number of levels of convolutional-pooling layer pairs, number of feature maps (channels) within convolutional layers, order of Chebyshev polynomial used within convolutional layers, number of neurons in the last layer, and dropout rate. For this purpose, three projects with different average number of AST nodes were selected. Sorted by the average number of AST nodes, these projects are as follows: *camel*, *lucene*, and *xalan*. For each of the selected projects, a 10-fold cross validation was performed with the earlier version of the project. Together with the parameters of the model, the optimal values for the pair of batch size and learning rate were explored.

A grid search has been performed over the space of all the parameters, except for the order of Chebyshev polynomials and batch size, which have been fixed at 3 and 4, respectively, due to the limited experimental environment. For each point in the parameters grid, the models with 1, 2, 3, and 4 levels of convolutional-pooling layer pairs have been trained over 50 epochs. The first convolutional layer of each model had the number of channels equal to the maximal number of nodes in ASTs from the project being analyzed, while each subsequent layer has a half of its previous layer channels. The number of neurons in the last hidden layer is set to a half of the size of a graph generated by the last pooling layer. The other parameters have been varied over the following range: dropout rate, $\{0.1, 0.3, 0.5\}$, and learning rate, $\{10^{-7}, 10^{-6}, 10^{-5}\}$. The average AUC score after running 10-fold cross validation of the selected project's models is shown in Figure 3, with point size and color for the dropout and learning rate, respectively.

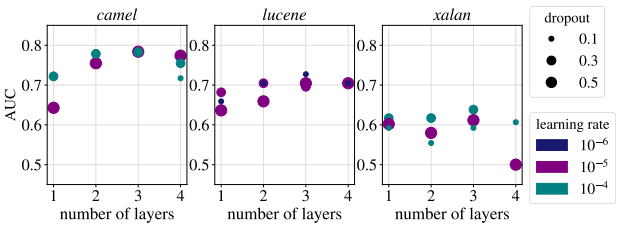


FIGURE 3. Results of hyperparameter tuning on the selected PROMISE projects.

As can be seen from Figure 3, a model whose architecture has three levels of convolutional-pooling layer pairs proved to be the most suitable for each of the three selected projects. Moreover, it can be observed that applying a dropout of 0.3 after each level can improve the performance of a model trained for a PROMISE project with one of the largest average number of AST nodes, as well as that a larger dropout rate, i.e. 0.5, yielded better results than smaller rates for a model developed for a project with the smallest average number of AST nodes. Based on these considerations, it seems reasonable to develop a model whose architecture has three levels of convolutional-pooling layer pairs for each PROMISE project. Furthermore, since it has been shown that using dropout of 0.3 can be helpful in training a model for projects with a relatively large average number of

AST's nodes, we set the dropout for each project to this value, except for *camel* project where a dropout of 0.5 is set, as suggested by results and having in mind it has a relatively smaller average number of nodes than other projects being analyzed. The optimal learning rate was searched for within the range listed above for each PROMISE project because we did not observe the regularity of its value and the average number of AST nodes within this hyperparameter tuning.

D. BASELINE METHODS

To answer RQ1, we compared the proposed model with the logistic regression classifier (LRC) and the random forest classifier (RFC), both of which are widely used for defect prediction of software modules based on traditional code features describing these modules [37]–[40].

Answering RQ2 requires comparing the proposed model with the *state-of-the-art* AST-based approaches. However, to ensure a fair comparison between the approaches, the risk of errors in reproducing the results of the approaches should be avoided. Therefore, we select the approaches that provide results on all seven PROMISE projects used in this experiment as baselines. These approaches are the following:

- DBN [3]: This approach uses a Deep Belief Network to automatically learn semantic features by using vectors representing tokens extracted of the AST from the source code.
- CNN [5]: This is a defect prediction model based on a neural network with three convolutional layers and four dense layers. It takes an integer vector representing string tokens of an AST as input.
- SEML [9]: This is an LSTM-based model for defect prediction. It can automatically learn the semantic information of a source code from a sequence of nodes of a corresponding AST.
- MPT [6]: This approach to defect prediction uses multi-perspective tree embedding to learn the representation an AST in an unsupervised manner.
- DP-T [11]: This model classifies a source code as defective or non-defective based on a representation of the sequence of nodes from the corresponding AST generated by Transformer framework.
- CSEM [13] The model is a cascade of a GNN that extracts features from AST and CFG of the source code in an unsupervised manner, and a logistic regression classifier that is trained for the task of defect prediction using such extracted features.

Although Zhao *et al.* [13] provided results from CSEM for six out of seven PROMISE projects, we believe it is important to perform this comparison as it allows us to assess whether an end-to-end GNN-based model (ours) is more successful in predicting defective source code than a traditional classifier that uses features extracted from a GNN-based model trained in an unsupervised manner (CSEM).

E. EXPERIMENTAL RESULTS

This subsection addresses the research questions. It describes the experiments conducted and their results, which are used to answer the questions.

1) PERFORMANCE COMPARISON OF THE PROPOSED MODEL AND TRADITIONAL CLASSIFIERS TRAINED USING TRADITIONAL FEATURES (RQ1)

To answer this question, we compared the performance of LRC and RFC trained on traditional code features representing modules from PROMISE projects with the performance of the proposed model (DP-GCNN) trained on ASTs from modules' source files.

The hyperparameters of the DP-GCNN were tuned before training, as described in Section IV-C. Similarly, using 10-fold cross-validation while monitoring the average F-score on PROMISE projects, the `solver`, `penalty`, and `C` hyperparameters of LRC and the `max_depth` and `n_estimators` hyperparameters of RFC were tuned by randomly examining the combinations of the values of these hyperparameters within a set of ranges typical of each of them. The values of the other hyperparameters of RFC and LRC were set to the default values provided by scikit-learn. Moreover, class weights were used in training all three models to prevent overfitting and improve performance [41]. In particular, the class weights were set to a value inversely proportional to their respective frequencies. Finally, given the large inconsistency of prediction results between existing software defect prediction models [42], we decided to repeat our experiment 30 times to reduce the likelihood of errors and thus obtain a more reliable experiment.

The results of AUC and F-score performance measures are shown in Table 3, with the best results among classifiers highlighted in bold. Additionally, obtained scores are shown as the violin plots in Figure 4. Table 4 shows the average AUC and F-scores obtained by the classifiers in the analyzed projects.

It can be seen from both the violin plots and numerical values that the proposed model DP-GCNN was more successful than the traditional classifiers LRC and RFC in predicting defective software modules from PROMISE projects in terms of both AUC and F-score. Although LRC achieved the same AUC score as DP-GCNN on *lucene* project, LRC was outperformed by DP-GCNN on the same project by 16% in terms of F-score. Moreover, DP-GCNN achieved 5% better mean AUC score (0.68 vs. 0.63) and 10% better mean F-score (0.61 vs. 0.51) than LRC on PROMISE projects. It can be concluded that DP-GCNN generally performed better than LRC for PROMISE data set, especially for *synapse* and *xalan* projects, where it outperformed LRC by 9% in terms of AUC score and by 14% and 12% in terms of F-score, respectively. Similarly, DP-GCNN outperformed RFC by 7% in terms of mean AUC score (0.68 vs. 0.61) and by 13% in terms of average F-score (0.61 vs. 0.48) on PROMISE projects. It also outperformed RFC in terms of AUC and F-score for every project from PROMISE data set.

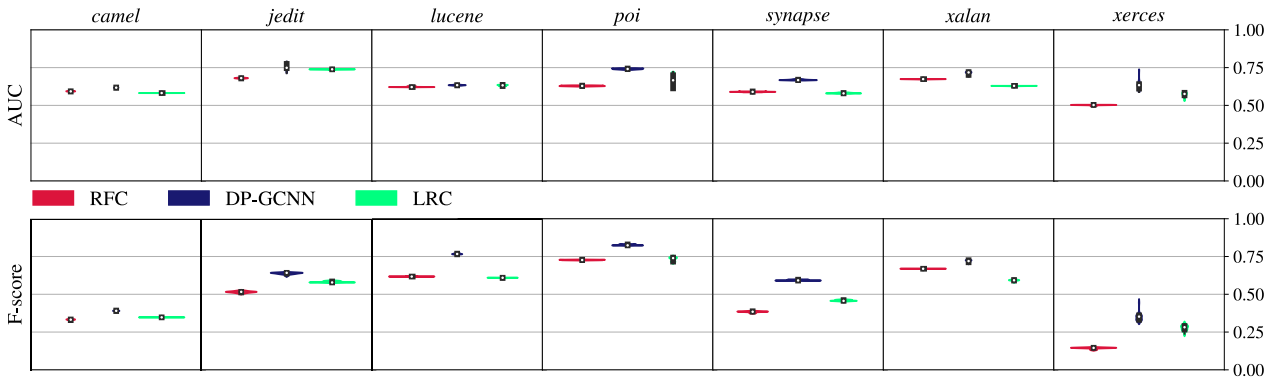


FIGURE 4. Violin plots of AUC and F-scores achieved by RFC, DP-GCNN nd RFC across 30 repetitions of the experiment.

TABLE 3. Performance comparison in terms of the AUC and F-score of the proposed model (DP-GCNN), the random forest classifier (RFC), and the logistic regression classifier (LRC) on seven projects from PROMISE data set.

	AUC		
Project	**DP-GCNN**	**RFC**	**LRC**
camel	$\mathbf{0.62 \pm 0.00}$	0.59 ± 0.00	0.58 ± 0.00
jedit	$\mathbf{0.76 \pm 0.02}$	0.68 ± 0.00	0.74 ± 0.01
lucene	$\mathbf{0.63 \pm 0.00}$	0.62 ± 0.00	$\mathbf{0.63 \pm 0.00}$
poi	$\mathbf{0.74 \pm 0.00}$	0.63 ± 0.00	0.65 ± 0.00
synapse	$\mathbf{0.67 \pm 0.00}$	0.59 ± 0.00	0.58 ± 0.01
xalan	$\mathbf{0.71 \pm 0.01}$	0.67 ± 0.00	0.62 ± 0.01
xerces	$\mathbf{0.63 \pm 0.03}$	0.50 ± 0.00	0.57 ± 0.02

	F-score		
Project	**DP-GCNN**	**RFC**	**LRC**
camel	$\mathbf{0.39 \pm 0.00}$	0.33 ± 0.00	0.35 ± 0.00
jedit	$\mathbf{0.64 \pm 0.01}$	0.52 ± 0.01	0.58 ± 0.01
lucene	$\mathbf{0.77 \pm 0.00}$	0.62 ± 0.00	0.61 ± 0.01
poi	$\mathbf{0.83 \pm 0.00}$	0.73 ± 0.00	0.73 ± 0.02
synapse	$\mathbf{0.59 \pm 0.00}$	0.39 ± 0.00	0.45 ± 0.01
xalan	$\mathbf{0.72 \pm 0.01}$	0.67 ± 0.00	0.60 ± 0.01
xerces	$\mathbf{0.35 \pm 0.02}$	0.14 ± 0.00	0.27 ± 0.02

TABLE 4. Average AUC and F-score of the proposed model (DP-GCNN), the random forest classifier (RFC), and the logistic regression classifier (LRC) for seven PROMISE projects.

	DP-GCNN	**RFC**	**LRC**
AUC	**0.68**	0.61	0.63
F-score	**0.61**	0.48	0.51

The largest performance difference between these two models is observed in *xerces* project, where DP-GCNN achieved 13% higher AUC and 21% higher F-score than RFC. In addition, compared to LRC and RFC, the DP-GCNN performed better on both measures for *camel* and *xerces* projects, which suffer the most from the class imbalance problem in the analyzed data set.

In order to ensure the credibility of the results, the statistical significance of the performance differences in terms of AUC and F-score of DP-GCNN, LRC, and RFC for the data set PROMISE should be evaluated. For this purpose, the performances of these classifiers are compared using the Friedman test [43], a non-parametric, multiple comparison tests for related samples. The null-hypothesis of the test states that there is no difference in the performance of the compared classifiers based on their ranks. If the p-value obtained by the test is less than or equal to the level of significance α of the test, then the null-hypothesis is rejected. In our experiment, α was set at 0.05.

The Friedman test is performed separately for both performance measures, with each classifier represented by a list of average scores it obtained on each of the seven PROMISE projects. The classifiers are first ranked for each project separately, and then their average ranks R are compared, with the lowest value R being the best. The ranks are shown in Table 5, where R_{AUC} and $R_{\text{F-score}}$ represents the average rank of the classifiers when compared based on AUC and F-score, respectively.

TABLE 5. Rankings obtained by the Friedman test of the proposed model (DP-GCNN), the random forest classifier (RFC), and the logistic regression classifier (LRC) when they are compared based on AUC (R_{AUC}) and F-score ($R_{\text{F-score}}$).

Model	R_{AUC}	$R_{\text{F-score}}$
DP-GCNN	**1.071429**	**1.000000**
LRC	2.357143	2.357143
RFC	2.571429	2.642857

For the rank comparison based on the AUC and F-scores obtained by the classifiers, the Friedman test yielded a p-value of 0.008415 and 0.003725, respectively. These results indicate that the null-hypothesis is rejected for both performance measures, so a further statistical analysis is required. The analysis involves the use of an appropriate post-hoc statistical test to determine which of the compared classifiers has these differences. Since the power of such a test is much greater if all classifiers are compared to only one, proposed classifier and not between themselves [44], we carried out a step-down many-to-one comparison procedure proposed by Holm [45]. In this context, we compared the best ranked

classifier, i.e. DP-GCNN, with other classifiers by testing the hypotheses in the form ''There is no difference in the performance of DP-GCNN and CLASSIFIER in terms of the measure.'' for a CLASSIFIER $\in$ {LRC, RFC}. The test is performed separately for each performance measure, with the level of significance α set at 0.05.

Post-hoc analysis using the Holm's step-down procedure begins with ranking the hypotheses by their p-values, so that the hypothesis with the smallest p-value is ranked highest. Starting with the highest ranked hypothesis, each of these p-values is compared to the significance level α divided by a rank of hypothesis i. If a p-value is below such adjusted significance level, the corresponding hypothesis is rejected and the second hypothesis can be tested. The test continues until a particular hypothesis cannot be rejected. At that moment, all other hypotheses are also retained. The results of the Holm test are shown in Table 6, where the column ''Model'' refers to a classification model whose performance is compared to the performance of DP-GCNN.

TABLE 6. The results of the Holm test to evaluate the statistical significance of the differences in AUC and F-score obtained from DP-GCNN and each of the random forest classifier (RFC), and the logistic regression classifier (LRC).

		AUC		F-score	
i	α/i	Model	p-value	Model	p-value
1	0.05	LRC	0.0161569	LRC	0.0111176
2	0.025	RFC	0.0050123	RFC	0.0021156

As can be seen from the Table 6, the p-values obtained are below the corresponding adjusted significance levels, so both hypotheses are rejected, which means DP-GCNN performs significantly better than LRC and RFC.

From the comparison, it can be concluded that the performance of the proposed model is better than LRC and RFC in terms of both AUC and F-score for PROMISE projects.

2) PERFORMANCE COMPARISON OF THE PROPOSED MODEL AND THE *STATE-OF-THE-ART* APPROACHES (RQ2)

In this experiment we investigate whether the performance of the proposed model is comparable to the reported performances of the *state-of-the-art* AST-based approaches, listed in Section IV-D. To this end, the proposed model was trained using early stopping technique as described in IV and a weighted loss function, with class weights set as in the experiment analyzed in Section IV-E1. The hyperparameters of the model were set to optimal values found as described in Section IV-C, while the learning rate and batch size were selected separately for each project. In particular, for each PROMISE project, the learning rate was set to an appropriate value so that the learning process converges within the iteration limit with the biggest batch size that fits in memory.

The results of the models with which we compare our model come from the corresponding studies. However, as suggested in previous statistical studies [46]–[48], an experiment should be repeated at least 30 times to minimize the statistical bias and variance of the results; otherwise, the risk of obtaining non-repeatable results and drawing wrong conclusions increases [49]. Despite this criterion, all the studies we compare with, except the study conducted by Pan *et al.* [5], repeated their experiments less than 30 times. Although we are aware of the risk that comparing performance is not completely reliable because the reported performances of these models cannot be considered statistically significant, we make the comparison with more than one *state-of-the-art* model, thus allowing a better assessment of the performance of the proposed model. The result of this comparison is shown in Figure 5, where the numbers in parentheses refer to the number of repetitions of each experiment involving model training and testing.

Since the result for CSEM on the *xalan* project is not reported, the only scientifically correct way to compare the performance of the models on the PROMISE data set is to make two comparisons. In the first comparison, the models that provide results for all PROMISE projects are compared based on their average score for all PROMISE projects. In the second comparison, all models are compared, but their average scores are calculated considering all PROMISE projects except for the *xalan* projects. The comparisons are shown in Figure 6.

Considering the approaches whose results were based on less than 30 repetitions of an experiment and reported results for all seven PROMISE projects, DP-GCNN was 3.6% and 2.3% better than DBN and SEML, respectively, in terms of average F-score. In addition, out of 7 PROMISE projects analyzed, DP-GCNN was better than DBN and SEML on 6 projects. The largest improvement was in *lucene* project where it outperformed DBN by 11.9% and SEML by 8.6%. Furthermore, looking at the average F-score, DP-GCNN performed slightly worse than DP-T and MPT. More precisely, DP-T and MPT achieved a 1.3% and 1.5% higher average F-score than DP-GCNN, respectively. The largest performance differences were observed in the *camel* project, where DP-T and MPT performed better than DP-GCNN by 13.6% and 12.8%, respectively. However, among the analyzed models, the highest reported F-score for the *camel* project is 0.526 (DP-T), which means that the models generally provide low results for this project, which may be due to the presence of high class imbalance. Nevertheless, based on similar results for other PROMISE projects, and on average, the performance of DP-GCNN can be considered comparable to that of DBN, SEML, DP-T and MPT.

The only statistically significant comparison can be made between DP-GCNN and CNN, for which the authors report the average F-score achieved by the CNN in 30 repetitions of the training and testing process for all seven PROMISE projects. As can be seen in Figure 5, a large difference can be observed between the performance of DP-GCNN and CNN on *poi* project, with DP-GCNN outperforming CNN by 38.6%. Moreover, DP-GCNN outperformed CNN by 6.9% and 5% for *lucene* and *jedit* projects, respectively. On the other hand, for the projects *camel*, *synapse*, and *xalan*, CNN

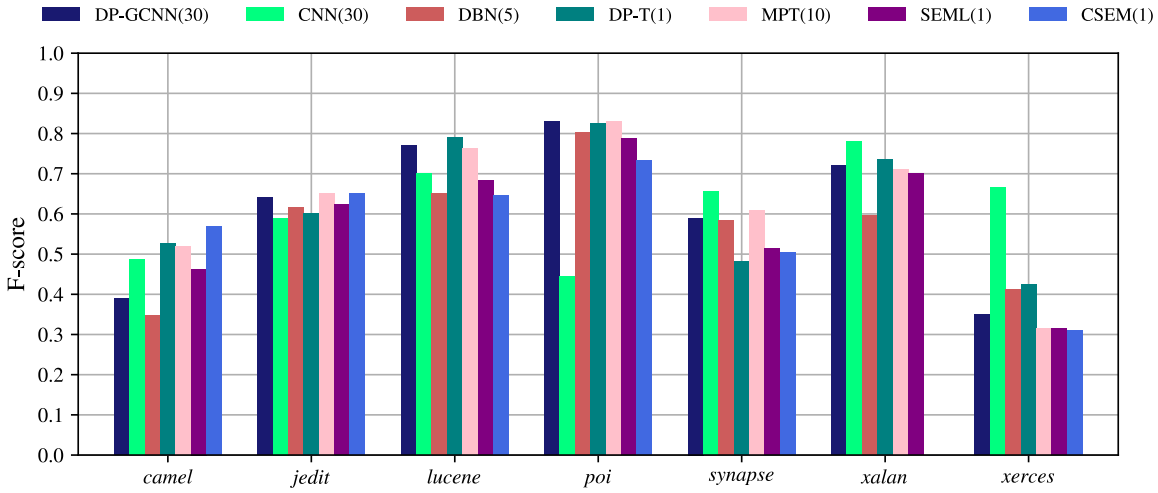


FIGURE 5. Performance comparison of DP-GCNN, DBN [3], CNN [5], SEML [9], MPT [6], DP-T [11], and CSEM [13], in terms of F-score on projects from PROMISE data set. The result for CSEM on the *xalan* project is not reported and therefore not included.

performed better than DP-GCNN by 9.7%, 6.5%, and 6%, respectively. For *xerces* project, CNN achieved a significantly higher F-score than DP-GCNN. However, none of the models analyzed were nearly as successful as CNN for this project. In particular, leaving aside the statistical significance, the model with the second best performance on *xerces*, DP-T, was outperformed by CNN by 24.2%. This difference is by far the largest over all differences in F-scores achieved by all models on other projects. Despite the large difference in performance for the *xerces* project, CNN performed only slightly better with a 0.5% higher average F-score, which is presented on the left in Figure 6. In summary, therefore, it can be stated that DP-GCNN provides comparable results to CNN on the PROMISE data set.

Based on the F-score for all PROMISE projects except for the *xalan* projects, DP-GCNN outperformed CSEM on five of six PROMISE projects. For the *camel* project, CSEM achieved an F-score of 0.57, which is the best result of all the models analyzed in this experiment, as shown in Figure 5. However, DP-GCNN performed better than CSEM with a 4% higher average F-score (0.61 vs. 0.57) for all six PROMISE projects. As can be seen in the Figure 6, CSEM and DBN performed the worst for all PROMISE projects except *xalan* project with an average F-score of 0.568. Compared to other models, i.e. SEML (0.565), MPT (0.614), CNN (0.591), and DP-T (0.608), DP-GCNN (0.595) achieved the second best result in terms of F-score.

In the present case, and considering the highest F-score for the *poi* project across all analyzed models, we can conclude that DP-GCNN provides comparable results to *state-of-the-art* models on the PROMISE data set. Although we cannot say with certainty that feature extraction by GNN-based models in an unsupervised manner would not give as good results as an end-to-end model like DP-GCNN on a different data set, the results obtained in this experiment support the fact that learning supervised GNN-based models can give better

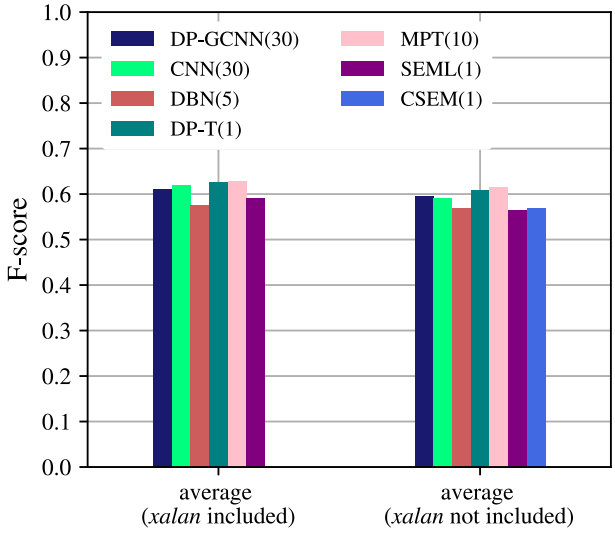


FIGURE 6. Performance comparison of DP-GCNN, DBN [3], CNN [5], SEML [9], MPT [6], DP-T [11], and CSEM [13] in terms of average F-score on PROMISE data set.

results than learning classifiers based on features extracted by GNN-based models in an unsupervised manner.

V. RELATED WORK

The existing work on identifying defective software modules in developing projects are discussed in Section V-A. Section V-B presents the most relevant existing models that classify software modules based on information from ASTs generated from the modules' source code.

A. IDENTIFICATION OF DEFECTIVE SOFTWARE MODULES

Models for identifying defect-prone project modules can be developed using the data of defective modules from the previous project version (WPDP) or from some other software project (CPDP). In both cases, features for representing project modules are first extracted from the data.

Depending on the project data, modules can be represented by process features, code features or a specific combination of these features, which is found to be beneficial for SDP models [50], [51]. Process features, which can be divided into developer metrics, code change metrics and development process metrics [27], have proven to be good indicators of software defects across many studies [26], [50]–[54]. Among the existing code features, most commonly used [1], [4], [34], [55]–[59] are static code and object-oriented metrics, which include Chidamber and Kemerer's metrics [60], McCabe's complexity measures [61], QMOOD metrics [62], coupling metrics [63], quality-oriented metrics [64], LCOM3 metric [65], Lines of Code (LOC), etc. In addition, network metrics, which reflect the interactions between software modules, have been used as features for representing software modules [7], [66]–[68]. However, in order to integrate the syntactical and semantic information of a source code into features as well, recent studies have directed their efforts at extracting code features from the source code represented by ASTs [3]–[8], [11] or CFGs [69], thereby leveraging deep learning-based models. Although the syntactic structure and semantic information of a source code can be captured by both AST and CFG, the latter has been rarely used, probably because it ignores the block structure of the code, which can be advantageous when integrated into code features.

The features, along with the corresponding defect labels, are fed as an input to a learning algorithm, which is used to build a model. Various SDP models have been proposed so far, with the most common being Logistic Regression classifier [3], [4], [8], [10], Decision Tree-based classifiers [39], [70], Naïve Bayes-based classifiers [25], [71], [72], and Support Vector Machine classifier [73], [74].

B. DEFECT PREDICTION USING ABSTRACT SYNTAX TREES

In recent times, researchers have been taking advantage of deep learning techniques, since such techniques learn features from the data directly, without the need for features to be constructed manually. In the existing studies, these techniques are usually applied for extracting relevant features from software's source code represented in the form of an AST.

A great majority of the existing studies have leveraged deep learning methods to extract features from linear sequences of AST's nodes that represent the source code of a software module. For instance, Wang *et al.* [3] have shown that a Deep Belief Network can automatically learn semantic features from tokens encoded into numerical vectors, which can be further used instead of traditional code features to yield better classification results. Moreover, Lie *et al.* [4] have applied a Convolutional Neural Network for effective feature generation from AST's token vectors, which are first embedded to real-valued vectors of fixed size. They combined the features thus obtained with traditional code features, and showed that such approach performs better in WPDP than some of the existing approaches on open-source projects. A similar, but more complex CNN architecture has been proposed by Pan *et al.* [5], who have reported that the proposed model outperforms *state-of-the-art* models in some evaluation metrics. Furthermore, Fan *et al.* [10] have proposed a model that first encodes the sequence of AST tokens into a module embedding. The embedding is then forwarded to a bidirectional Long Short-Term Memory (LSTM) network with attention mechanism to capture crucial features, which are then given as an input to a logistic regression classifier. Another complex deep model has been proposed by Zhang *et al.* [11]. They first select representative nodes from ASTs to form token vectors, which are converted into numerical vectors using a word embedding technique. The vectors are sent to an encoder-decoder-based model called transformer, which automatically extracts syntactic and semantic features. The extracted features are used to train a logistic regression classifier.

Only in a few work the structural information between AST's nodes represented by tokens has been considered. Specifically, in a study by Dam *et al.* [8], a tree-based network of LSTM units has been leveraged to obtain a vector representation of each AST's token. Using such vectors, they have generated a representation of the whole AST, which has shown promising results when used with the corresponding defect label to build a traditional classifier for identifying defective modules. Another unsupervised method for learning representation of the source code from an AST has been proposed by Shi *et al.* [6]. The method, called MPT-embedding, parses the nodes of ASTs for multiple perspectives and encodes the structural information of a tree into a vector sequence. The vector sequence representing a software module and the corresponding defect label are used to train a CNN-based classifier, whose results on open-source projects indicate that MPT-embeddings are of comparable quality to the features for representing software modules used in *state-of-the-art* approaches. Similarly, Zhao *et al.* [13] recently presented an unsupervised method for learning the contextual semantics of source code via hierarchical dependency structures and graph attention networks. The method extracts features to represent a software module from both AST and CFG from the module's source code. The features are used along with the corresponding defect label to train logistic regression classifiers for three tasks, including defect prediction. Their approach outperformed baseline methods in open-source projects on all tasks analyzed.

In contrast to these studies, Xu *et al.* [12] proposed an end-to-end GCNN-based model that learns the latent defective representation of software modules from code snippets, project information, and fix-inducing changes. Nodes of the part of the AST are enriched with the concept features, which reflect both the semantic information from the source code represented by the AST and the information extracted from the analyzed project and fix-inducing code changes. The experimental results on modules from various Java projects showed that their approach led to better defect prediction performance compared with baseline and *state-of-the-art* approaches. Unlike the above-mentioned approaches,

we propose an end-to-end model which is trained in a supervised manner using the whole AST representing the software module. Our model utilizes a powerful GCNN whose architecture is adjusted to each projects modules,allowing the model to capture the same amount of information regardless of the modules size. In this way, a complete information from modules is preserved and can be utilized for the model development.

VI. THREATS TO VALIDITY

As with any empirical study, there are threats to validity that should be discussed. Below we discuss the external, internal, construct and conclusion validity of our study.

A. EXTERNAL VALIDITY

External validity focuses on examining whether and to what extent a generalization of the research results is possible. The quality of the experimental results depends on the data set used, and therefore we have chosen to use the data set commonly used in software defect prediction studies. As such, it should be suitable for developing and validating models for identifying defect-prone software modules. However, the experiments conducted in this research can also be performed with a different data set.

B. INTERNAL VALIDITY

To assess the advantage of using the proposed model for predicting defective software modules, it should be compared with the commonly used SDP models. For this purpose, we had to decide against which SDP models the proposed model should be compared. Even though our decisions are made according to the common practice in SDP and are also justified in this paper, it is possible that the internal validity of our study is influenced by the preference of the models chosen. In order to minimize this possibility, we have decided to use different but commonly used SDP models in this research. In future research, however, more different SDP models can be used for the comparison with the proposed model.

C. CONSTRUCT VALIDITY

The evaluation of the proposed model depends directly on the measures used to assess the performance. To make the assessment fair, we reported various performance measures, including those used in the majority of SDP studies. Nevertheless, other appropriate evaluation metrics may also be used for evaluation purposes.

D. CONCLUSION VALIDITY

To test the validity of the conclusions drawn in this study, we ensured the statistical significance of our results by repeating each experiment 30 times, thus fulfilling a requirement of the central limit theorem. To make the research results more reliable, we also conducted appropriate significance tests to show that the results of the model are significantly different from the results of traditional SDP models. For this purpose, we used the Friedman test and Holm's step-down procedure, but it is also possible to perform other statistical tests as long as the data meet the requirements of the tests. Finally, considering the insufficient number of experiment repetitions in the results of the *state-of-the-art* approaches, an indicative comparative evaluation was only possible in the context of this work.

VII. CONCLUSION AND FUTURE WORK

With the rapid development of ever larger and more complex software systems comes the need for quick and accurate methods for detecting potential defects in source code of software. Various models have been proposed for this task, with deep learning models that analyze ASTs from source code as the most successful so far. However, many of such models are not specialized for tree-structured data such as ASTs, or take only partial information of ASTs being analyzed.

In this work, we present DP-GCNN, a defect prediction model based on a neural network architecture that is specifically tailored for graph data, which ASTs belong to. The architecture of the proposed DP-GCNN is adaptive, meaning its structure can be adapted to the software to be analyzed, which ensures the DP-GCNN is able to process the modules of different projects with the same level of detail, regardless of the software size. The experiments conducted have shown that DP-GCNN outperforms traditional SDP models based on static code features in terms of AUC and F-score for projects from the PROMISE data set. Compared to the F-score reported by the *state-of-the-art* SDP models based on AST, DP-GCNN has demonstrated comparable predictive capabilities for PROMISE projects.

Future work will focus on investigating the applicability of DP-GCNN and any modification thereof in predicting fault-prone software modules for source code files of different sizes from the file analyzed in this paper and for software modules written in other programming languages.

ACKNOWLEDGMENT

The Titan X Pascal used for this research was donated by NVIDIA Corporation.

REFERENCES

[1] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: Current results, limitations, new approaches," *Autom. Softw. Eng.*, vol. 17, no. 2, pp. 375–407, Dec. 2010.

[2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, Feb. 2015.

[3] 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.

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

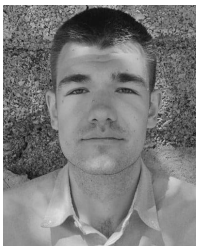
[5] C. Pan, M. Lu, B. Xu, and H. Gao, "An improved CNN model for within-project software defect prediction," *Appl. Sci.*, vol. 9, no. 10, p. 2138, May 2019.

[6] K. Shi, Y. Lu, G. Liu, Z. Wei, and J. Chang, "MPT-embedding: An unsupervised representation learning of code for software defect prediction," *J. Softw., Evol. Process.*, vol. 33, no. 4, p. e2330, Apr. 2021.

[7] S. Meilong, P. He, H. Xiao, H. Li, and C. Zeng, "An approach to semantic and structural features learning for software defect prediction," *Math. Problems Eng.*, vol. 2020, Apr. 2020, Art. no. 6038619.
[8] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, "Lessons learned from using a deep tree-based model for software defect prediction in practice," in *Proc. IEEE/ACM 16th Int. Conf. Mining Softw. Repositories (MSR)*, Oct. 2019, pp. 46–57.
[9] H. Liang, Y. Yu, L. Jiang, and Z. Xie, "Seml: A semantic LSTM model for software defect prediction," *IEEE Access*, vol. 7, pp. 83812–83824, 2019.
[10] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, "Software defect prediction via attention-based recurrent neural network," *Sci. Program.*, vol. 2019, Apr. 2019, Art. no. 6230953.
[11] Q. Zhang and B. Wu, "Software defect prediction via transformer," in *Proc. IEEE 4th Inf. Technol., Netw., Electron. Autom. Control Conf. (ITNEC)*, Jun. 2020, pp. 874–879.
[12] J. Xu, F. Wang, and J. Ai, "Defect prediction with semantics and context features of codes based on graph representation learning," *IEEE Trans. Rel.*, vol. 70, no. 2, pp. 613–625, Jun. 2021.
[13] Z. Zhao, B. Yang, G. Li, H. Liu, and Z. Jin, "Precise learning of source code contextual semantics via hierarchical dependence structure and graph attention networks," *J. Syst. Softw.*, vol. 184, Feb. 2022, Art. no. 111108.
[14] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 783–794.
[15] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proc. AAAI Conf. Artif. Intell.*, 2016, pp. 1287–1293.
[16] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Aug. 2016, pp. 87–98.
[17] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Feb. 2020, pp. 261–271.
[18] J. Zhao, K. Xia, Y. Fu, and B. Cui, "An AST-based code plagiarism detection algorithm," in *Proc. 10th Int. Conf. Broadband Wireless Comput., Commun. Appl. (BWCCA)*, Oct. 2015, pp. 178–182.
[19] J. Feng, B. Cui, and K. Xia, "A code comparison algorithm based on ast for plagiarism detection," in *Proc. 4th Int. Conf. Emerg. Intell. Data Web Technol.*, Oct. 2013, pp. 393–397.
[20] A. Bogdanova, "Source code authorship attribution using file embeddings," in *Proc. ACM SIGPLAN Int. Conf. Syst., Program., Lang., Appl., Softw. Hum.*, Oct. 2021, pp. 31–33.
[21] F. Ullah, S. Jabbar, and F. Al-Turjman, "Programmers' de-anonymization using a hybrid approach of abstract syntax tree and deep learning," *Technol. Forecasting Social Change*, vol. 159, Oct. 2020, Art. no. 120186.
[22] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," 2017, *arXiv:1711.00740*.
[23] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu, "A general software defect-proneness prediction framework," *IEEE Trans. Softw. Eng.*, vol. 37, no. 3, pp. 356–370, Oct. 2010.
[24] G. Dong and H. Liu, "Feature engineering for machine learning and data analytics," in *Feature Generation and Engineering for Software Analytics*. Boca Raton, FL, USA: CRC Press, 2018, pp. 335–358.
[25] A. Okutan and O. T. Yıldız, "Software defect prediction using Bayesian networks," *Empirical Softw. Eng.*, vol. 19, no. 1, pp. 154–181, 2014.
[26] 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, 2014.
[27] L. Jiang, S. Jiang, L. Gong, Y. Dong, and Q. Yu, "Which process metrics are significantly important to change of defects in evolving projects: An empirical study," *IEEE Access*, vol. 8, pp. 93705–93722, 2020.
[28] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: Going beyond Euclidean data," *IEEE Signal Process. Mag.*, vol. 34, no. 4, pp. 18–42, Jul. 2017.
[29] I. Ronald Ward, J. Joyner, C. Lickfold, Y. Guo, and M. Bennamoun, "A practical tutorial on graph neural networks," 2020, *arXiv:2010.05234*.
[30] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," 2016, *arXiv:1606.09375*.
[31] F. M. Bianchi, D. Grattarola, and C. Alippi, "Spectral clustering with graph neural networks for graph pooling," in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 874–883.
[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.
[33] L. Prechelt, "Early stopping-but when?" in *Neural Networks: Tricks of the Trade*. Berlin, Germany: Springer, 1998, pp. 55–69.
[34] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proc. 6th Int. Conf. Predictive Models Softw. Eng.*, 2010, pp. 1–10.
[35] Y. Jiang, J. Lin, B. Cukic, and T. Menzies, "Variance analysis in software fault prediction models," in *Proc. 20th Int. Symp. Softw. Rel. Eng.*, 2009, pp. 99–108.
[36] G. A. Lujan-Moreno, P. R. Howard, O. G. Rojas, and D. C. Montgomery, "Design of experiments and response surface methodology to tune machine learning hyperparameters, with a random forest case-study," *Expert Syst. Appl.*, vol. 109, pp. 195–205, Oct. 2018.
[37] T. M. Khoshgoftaar and K. Gao, "Feature selection with imbalanced data for software defect prediction," in *Proc. Int. Conf. Mach. Learn. Appl.*, Oct. 2009, pp. 235–240.
[38] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, "Heterogeneous defect prediction," *IEEE Trans. Softw. Eng.*, vol. 44, no. 9, pp. 874–896, Sep. 2018.
[39] A. Kaur and R. Malhotra, "Application of random forest in predicting fault-prone classes," in *Proc. Int. Conf. Adv. Comput. Theory Eng.*, Dec. 2008, pp. 37–43.
[40] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust prediction of fault-proneness by random forests," in *Proc. 15th Int. Symp. Softw. Rel. Eng.*, 2004, pp. 417–428.
[41] J. M. Johnson and T. M. Khoshgoftaar, "Survey on deep learning with class imbalance," *J. Big Data*, vol. 6, no. 1, pp. 1–54, Dec. 2019.
[42] D. Bowes, T. Hall, and J. Petrić, "Software defect prediction: Do different classifiers find the same defects?" *Softw. Qual. J.*, vol. 26, no. 2, pp. 525–552, 2018.
[43] M. Friedman, "The use of ranks to avoid the assumption of normality implicit in the analysis of variance," *J. Amer. Statist. Assoc.*, vol. 32, no. 200, pp. 675–701, Dec. 1937.
[44] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *J. Mach. Learn. Res.*, vol. 7, pp. 1–30, Dec. 2006.
[45] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandin. J. Statist.*, vol. 4, pp. 65–70, Jan. 1979.
[46] J. A. Rice, *Mathematical Statistics and Data Analysis*. Boston, MA, USA: Cengage Learning, 2006.
[47] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proc. 33rd Int. Conf. Softw. Eng.*, May 2011, pp. 1–10.
[48] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 1–18, Jan. 2017.
[49] T. Kalibera and R. Jones, "Quantifying performance changes with effect size confidence intervals," 2020, *arXiv:2007.10899*.
[50] Q. He, B. Shen, and Y. Chen, "Software defect prediction using semi-supervised learning with change burst information," in *Proc. IEEE 40th Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, Jun. 2016, pp. 113–122.
[51] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, pp. 653–661, Jul. 2000.
[52] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Inf. Softw. Technol.*, vol. 55, no. 8, pp. 1397–1418, 2013.
[53] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Proc. 33rd Int. Conf. Softw. Eng.*, May 2011, pp. 481–490.
[54] G. R. Choudhary, S. Kumar, K. Kumar, A. Mishra, and C. Catal, "Empirical analysis of change metrics for software fault prediction," *Comput. Electr. Eng.*, vol. 67, pp. 15–24, Mar. 2018.
[55] L. C. Briand and J. Wüst, "Empirical studies of quality models in object-oriented systems," *Adv. Comput.*, vol. 56, pp. 97–166, Oct. 2002.
[56] M. O. Elish, A. H. Al-Yafei, and M. Al-Mulhem, "Empirical comparison of three metrics suites for fault prediction in packages of object-oriented systems: A case study of Eclipse," *Adv. Eng. Softw.*, vol. 42, no. 10, pp. 852–859, 2011.
[57] R. Shatnawi and W. Li, "The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process," *J. Syst. Softw.*, vol. 81, no. 11, pp. 1868–1882, 2008.
[58] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu, "Dictionary learning based software defect prediction," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 414–423.

[59] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, Jan. 2007.
[60] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.
[61] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.
[62] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, Jan. 2002.
[63] R. Martin, "OO design quality metrics," *Anal. Dependencies*, vol. 12, pp. 151–170, Oct. 1994.
[64] M.-H. Tang, M.-H. Kao, and M.-H. Chen, "An empirical study on object-oriented metrics," in *Proc. 6th Int. Softw. Metrics Symp.*, 1999, pp. 242–249.
[65] B. Henderson-Sellers, *Object-Oriented Metrics: Measures Complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, 1995.
[66] R. Premraj and K. Herzig, "Network versus code metrics to predict defects: A replication study," in *Proc. Int. Symp. Empirical Softw. Eng. Meas.*, Sep. 2011, pp. 215–224.
[67] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proc. 13th Int. Conf. Softw. Eng.*, 2008, pp. 531–540.
[68] P. He, B. Li, Y. Ma, and L. He, "Using software dependency to bug prediction," *Math. Problems Eng.*, vol. 2013, Oct. 2013, Art. no. 869356.
[69] A. Viet Phan, M. Le Nguyen, and L. Thu Bui, "Convolutional neural networks over control flow graphs for software defect prediction," in *Proc. IEEE 29th Int. Conf. Tools Artif. Intell. (ICTAI)*, Nov. 2017, pp. 45–52.
[70] A. G. Koru and H. Liu, "Building effective defect-prediction models in practice," *IEEE Softw.*, vol. 22, no. 6, pp. 23–29, Nov. 2005.
[71] R. T. Asmono, R. S. Wahono, and A. Syukur, "Absolute correlation weighted naïve Bayes for software defect prediction," *J. Softw. Eng.*, vol. 1, no. 1, pp. 38–45, 2015.
[72] 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.*, Apr. 2015, pp. 1644–1647.
[73] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *J. Syst. Softw.*, vol. 81, no. 5, pp. 649–660, 2008.
[74] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson, "Software defect prediction using static code metrics underestimates defect-proneness," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2010, pp. 1–7.

LUCIJA ŠIKIĆ received the master's degree in computer science from the University of Zagreb, in 2018. Her Ph.D. project is source code defect prediction. She is currently a Senior Researcher at the Consumer Computing Laboratory, Faculty of Electrical Engineering and Computing, University of Zagreb. She has published articles in IEEE Access and *Journal of Software and Systems*.



ADRIAN SATJA KURDIJA (Member, IEEE) received the Ph.D. degree in computer science from the Faculty of Electrical Engineering and Computing, University of Zagreb, in 2020. His Ph.D. project deals with service selection and QoS prediction. He is currently a Research Assistant at the Consumer Computing Laboratory, Faculty of Electrical Engineering and Computing, University of Zagreb. He has published articles in IEEE Communications Letters, *European Journal of Operational Research*, *International Journal of Web and Grid Services*, *Knowledge-Based Systems*, and IEEE Transactions on Services Computing.

KLEMO VLADIMIR (Member, IEEE) received the Ph.D. degree in computer science from the Faculty of Electrical Engineering and Computing, University of Zagreb, in 2013. He is currently an Assistant Professor at the Faculty of Electrical Engineering and Computing, University of Zagreb. He has published results of his research in multiple journals and conferences, including IEEE Transactions on Services Computing and *Knowledge-Based System*. His research interests include recommendation systems, data mining, and software engineering.

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