End-to-End Deep Learning for Defect Prediction

***Abstract*— The problem of software defect prediction, which involves identifying likely erroneous files in a computer program or system, has recently gained much attention in software engineering community. The ability to identify defects would help developers better focus their efforts on assuring software quality. Traditional approaches for defect prediction generally begin by a feature construction step to encode the characteristics of programs, followed by a defect modeling stage that involves training a classification algorithm. However, the feature construction stage in these approaches is carried out without considering known defect labels, potentially leading to suboptimal learned features. In light of this deficiency, we propose in this paper a new deep learning approach called *deep discriminative autoencoder* (DDA), which performs end-to-end training to jointly learn discriminative (latent) features and an accurate classification model for effective defect identification. Preliminary experimental results on four popular software projects show that our DDA approach significantly outperforms traditional methods on both within-project (WP) and cross-project (CP) defect prediction. In particular, our approach improves on average by 19.63% in terms of F1 score for the WP problems. For the CP problems, DDA outperforms other methods by 18.95% in terms of F1 score.**

I. Introduction

Software defect prediction techniques [8], [12], [44] have been developed to automatically detect defects among program elements, which in turn help developers reduce their testing efforts and minimize software development costs. In a defect prediction task, one typically constructs defect prediction models from software history data, and uses these models to predict whether new instances of code elements (e.g., files, changes, and methods) contain defects. Traditionally, research efforts to construct accurate defective prediction models fall into two directions: the first direction focuses on manually designing a set of discriminative features that can represent defects more effectively; the second direction aims to build a new machine learning algorithm that improves the conventional prediction models.

In the past, most researchers manually designed features to filter buggy source files from non-buggy files. Typically, features are constructed based on changes in source code (i.e., the number of lines of code added, removed, etc.), complexity of code, or understanding of source code [12], [5], [22], [4], [7]. A common drawback of these approaches is that the features constructed cannot adequately capture the contextual semantic meanings of different programs. For example, two Java programs may have identical `if` and `for` statements, except that the if statement is outside the for loop in the first program whereas the second program has the if statement inside the for loop. Although the two program files have different semantics, the features generated by the traditional approaches may be identical, failing to distinguish the semantics of the two programs. As such, the defect prediction models constructed based on these features are often suboptimal.

In view of this limitation, Wang et al. [40] recently developed a deep belief network (DBN) model [10] to automatically learn embedding features (referred to as semantic information) that are a compressed representation of token vectors extracted from a program's Abstract Syntax Tree (AST). The learned features are then utilized as training input to build a defect classification model. However, in this approach, the embedding features and defect prediction model are built separately. That is, the embedding features are learned from source files in an unsupervised manner, without considering the true label of the program element. Moreover, token values are mapped to unique integer identifiers without reflecting the importance of that token in the program element. Hence, the embedding features may be suboptimal for defect prediction purposes.

To address this shortcoming, we propose a new deep learning approach named *deep discriminative autoencoder* (DDA), which provides an end-to-end learning scheme to construct discriminative embedding features and accurate defect classification model in one go. DDA extends a deep autoencoder model [37], which is an unsupervised learning model. Our DDA adds a discriminative power to the deep autoencoder model, making it a supervised learning model. We summarize the key contributions of this paper below:

- We develop a new deep learning approach for defect prediction, which is trained end-to-end using a joint loss function that simultaneously takes into account the defect prediction quality and reconstruction quality of the embedding features.
- We conduct experiments on four popular Java software projects. The results show that our approach significantly improve traditional defect prediction methods by 8.6% and 5.4% in terms of F1 score, for within-project and cross project defect prediction tasks respectively.

The remainder of this paper is organized as follows. Section II elaborates our proposed DDA approach. Section III presents our experimental results, followed by discussion on threats to validity in Section III-E. Review of key related works is given in Section IV. Finally, Section V concludes this paper.

II. Proposed Approach

In this section, we explain how the input features are generated for DDA model and briefly present our proposed approach.

A. Parsing Source Code and Generating Input Features

Following Wang et al.'s approach [40], we extract a sequence of AST node tokens from source code files.

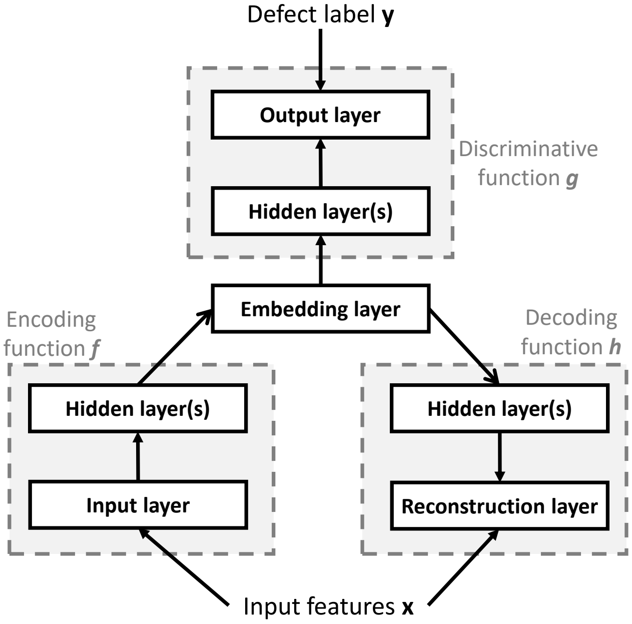

Fig. 1. Our proposed DDA model architecture

However, in contrast to Wang et al.'s approach which weights the extracted AST tokens as equally, we assign weights to the tokens using a term frequency–inverse document frequency (TF-IDF) scheme [20]. TF refers to the number of times a token appears in a source code file. IDF refers to the reciprocal of the number of source code files in the entire source code files that contain the token. TF-IDF of a token is a multiplication of its TF and IDF. The resultant sequence of AST tokens are weighted by their TF-IDF values.

B. Deep Discriminative Autoencoder

In this section, we describe our DDA approach to defect prediction, which aims to detect source code files that may potentially contain a bug. Firstly, let $\mathcal{X} = \{x_1, \ldots, x_i, \ldots, x_n\}$ denotes the set of source code files in a software project and $\mathcal{Y} = \{y_1, \ldots, y_i, \ldots, y_n\}$ represents the set of labels for the source code files, where n is the number of source code files in the project. A source code file is labelled as $y_i = 1$ if it contains a bug; otherwise, it is labelled as $y_i = 0$.

Unlike traditional approaches [41], [40], which learn embedding features and defect prediction model separately, our DDA approach performs an end-to-end learning to accomplish the two tasks in one shot. Specifically, DDA simultaneously learns three (non-linear) functions: 1) an *encoding function* f that maps input features to an embedding representation, 2) a *discriminative function* g that maps the embedding representation to defect class labels, and 3) a *decoding function* h that reconstructs the input features from the embedding representation. While an autoencoder model [37] only contains *encoding function* and *decoding function*, we add a *discriminative function* that maps embedding layer to output layer. Fig. 1 presents the architecture of our DDA model that realizes the three functions. Each function is represented using one or more hidden (fully-connected) layers. The embedding layer is shared by the three functions, while the input, output, and reconstruction layer is used by function f, g, and h.

To learn f, g and h simultaneously, we devise the following loss minimization problem:

$$\min \sum_{i=1}^{n} [\mathcal{L}_{discr}(g(f(x_i)), y_i) + \beta \mathcal{L}_{recon}(h(f(x_i)), x_i)] + \lambda \Omega(\theta) \tag{1}$$

where $\mathcal{L}_{discr}(g(f(x_i)), y_i)$ measures classification loss with respect to defect label y_i, $\mathcal{L}_{recon}(h(f(x_i)), x_i)$ is the reconstruction loss with respect to input feature x_i, and $\Omega(\theta)$ is the regularization terms for the set θ of all weight parameters within the DDA architecture. The parameters $\beta > 0$ and $\lambda > 0$ are user-defined, and serve to control the tradeoff between the different loss/regularization terms.

In this work, we define $\mathcal{L}_{discr}$ and $\mathcal{L}_{recon}$ respectively as:

$$\begin{aligned}\mathcal{L}_{discr}(g(f(x_i)), y_i) = & -[y_i \ln(\sigma(g(f(x_i)))) + \\ & (1 - y_i)\ln(1 - \sigma(g(f(x_i))))]\end{aligned} \tag{2}$$

$$\mathcal{L}_{recon}(h(f(x_i)), x_i) = \frac{1}{2}||h(f(x_i)) - x_i||^2 \tag{3}$$

while the regularization term Ω is given by:

$$\Omega(\theta) = \frac{1}{2}\sum_{w \in \theta} w^2 \tag{4}$$

where $\sigma(x) = \frac{1}{1+\exp(-x)}$ is the logistic/sigmoid function, and $w \in \theta$ is a particular weight parameter in the DDA network. It is worth noting that equation (2) corresponds the so-called cross-entropy loss commonly used for classification in deep learning [33], while equation (3) is the least square loss used to measure reconstruction quality in an autoencoder [37]. Finally, equation (4) corresponds to the ridge regularization term, which enforces the weight parameters w to be small so as to reduce the risk of data overfitting [3].

To minimize the joint loss function in (1), we employ in this work an adaptive gradient-based algorithm called Adam [16]. More specifically, Adam is an efficient algorithm for stochastic optimization that computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients. This method is straightforward to implement, is computationally efficient, and has little memory requirements [16], making it well-suited to optimize such deep architecture as our DDA model.

C. Handling Imbalanced Class Distribution

In defect prediction tasks, oftentimes there are only a handful of program files that contain bugs, while the other program files are clean (i.e., bug-free) [14]. As such, we can expect to see a highly-skewed (imbalanced) distribution of class labels (i.e., buggy vs clean). This imposes difficulties for gradient-based learning approaches, making them more biased towards the majority class (i.e., the class with more data instances). As such, class imbalance learning mechanisms would be helpful to tackle defect prediction problems [39].

In a similar vein, we develop a simple alternating (random) sampling strategy [18] when training DDA. In a nutshell, we divide the training data into two sets, i.e., buggy set and clean sets. Then, we perform an Adam update step by presenting a randomly-selected buggy sample and a randomly-selected clean sample in an alternating manner. That is, in update step i, we present a buggy sample to DDA, and a clean sample in update step $i + 1$, and so on. Effectively, this renders a balanced (bootstrapped) training data for DDA, which would help mitigate the bias from the majority (i.e., clean) class.

D. Parameter Setting for DDA Training

In this work, we use a DDA architecture with one hidden layer for each function f, g, and h. We configure the DDA model is as follows: The number of neurons (nodes) in the hidden layer of f and h is set to 1000, while that of g is 50. Also, the number of neurons in the embedding layer is set to 100. Meanwhile, the regularization parameter λ is set to 0.01, and the reconstruction parameter β is chosen via cross validation on the training data. Finally, we train DDA with a maximum training epoch of 75.

III. Experimental Results

We conduct extensive experiments to study the performance of the proposed approach and compare it with existing defect prediction approaches. We also discuss threats to the validity of our approach.

A. Evaluation Metrics

To measure defect prediction performance, we employ three different evaluation metrics: *Precision*, *Recall* and *F1* score. These metrics are widely used to evaluate the performance of defect prediction [23], [24], [29]. Below is the equation for each of these metrics:

$$Precision = \frac{TP}{TP + FP} \tag{5}$$

$$Recall = \frac{TP}{TP + FN} \tag{6}$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{7}$$

where *TP*, *FP*, and *FN* are considered as true positive, false positive, and false negative, respectively. True positive is the number of predicted defective files that are truly defective, while false positive is the number of predicted defective files that are actually not defective. False negative records the number of predicted non-defective files that are actually defective. A higher precision makes the manual inspection on a certain amount of predicted defective files find more defects, while an increase in recall reveals more defects in a project. F1 score considers both precision and recall.

B. Datasets

We perform several steps to create our benchmark dataset. We do not use PROMISE defect dataset[1] since the projects are old (i.e., average age is around 10 years). Firstly, we fetch the latest top open-source Java projects from GitHub (sorted by the number of their stars and forks). We ignore projects with less than 150 source files as these projects are too small to employ deep neural network. We also filter out projects which have less than 100 tested files. For our preliminary experiment, we pick 4 projects. For each project, we extract two versions: training version (i.e., version as of January 1^{st}, 2015), and testing version (i.e., version as of July 1^{st}, 2015).

For labeling training version, we extract commits between January 1^{st}, 2015 to July 1^{st} 2015. We then identify bug fixing commits by checking whether the commit message contains a bug fixing pattern. We follow the pattern used by Antoniol et al. [2] as follows.

$$\backslash bfix|\backslash bbug|\backslash bproblem|\backslash bdefect|\backslash bpatch$$

We consider changed files in bug fixing commits as buggy files and label their corresponding files (i.e., files of the same path) in training version as buggy. For labeling testing version, we extract commits between July 1^{st}, 2015 to January 1^{st} 2016 and perform the same labeling process that was done for the training version.

Table I shows statistics on this dataset. In average, our dataset contains around 783.88 source files with bug rate of 17.4%, showing the imbalanced problem in defect prediction [39], [14].

C. Baselines

We compare our approach with the defect prediction models constructed based on two traditional features. The first traditional features are embedding features generated following Wang et al. [40]. The second traditional features are AST features extracted from source code's AST. Specifically, we collect AST nodes from source code and represent the source code as a vector of term frequencies of the AST nodes. These two baselines were shown their effectiveness in solving defect prediction problem [40].

We employ three popular machine learning algorithms to build defect prediction models for each traditional features. These algorithms are widely used in software engineering [40], [39], [13] described as follows:

- Decision tree is used to build a tree-based classification model where branch nodes represent an option on feature values while leaf nodes represent predicted values [34].
- Logistic regression is a well-known classification model is employed in various application such as: health, statistics, data analysis, etc. [11].
- Naïve Bayes classifier, which is highly scalable, is a simple probabilistic classifiers based on applying Bayes' theorem [36].

[1] http://openscience.us/repo/defect/

TABLE I
DESCRIPTION OF FOUR POPULAR SOFTWARE PROJECTS.

Project	Description	Avg File	Avg Bug (%)
Checkstyle	a program to check whether source code conforms to coding standard	433.5	30.9
NuvolaBase	an add on to create, share, and exchange database in the cloud	1292.5	12.3
OrientDB	a Multi-Model DBMS with document and graphe engine	1194.5	9.17
Traccar	a server for various GPS tracking systems	215	17.3

TABLE II
PRECISION, RECALL AND F1 SCORES OF WITHIN-PROJECT PREDICTION. ALL THE SCORES ARE MEASURED BY PERCENTAGE. THE BEST F1 SCORES ARE HIGHLIGHTED IN BOLD. DEEP DISCRIMINATIVE AUTOENCODER, DECISION TREE, LOGISTIC REGRESSION, AND NAÏVE BAYES ARE DENOTED AS DDA, DT, LR, NB RESPECTIVELY.

Project	DDA	Embedding			AST		
		DT	LR	NB	DT	LR	NB
	P R F1	P R F1	P R F1	P R F1	P R F1	P R F1	P R F1
Checkstyle	74.6 84.7 **79.3**	78.3 55.3 64.8	83.6 68.8 75.5	72.3 82.9 77.3	82.4 63.5 71.8	79.7 62.4 70.0	82.7 64.7 72.6
Nuvolabase	33.8 66.0 **44.7**	32.8 7.51 12.2	36.2 44.7 40.0	31.1 73.5 43.7	50.7 14.2 22.2	59.6 12.2 20.3	20.3 41.1 27.1
OrientDB	32.9 47.9 **39.0**	25.6 6.94 10.9	27.3 47.2 34.6	20.8 69.4 32.0	40.3 18.8 25.6	44.2 13.2 20.3	12.1 38.2 18.4
Traccar	33.9 75.0 **46.7**	14.0 80.0 23.9	12.7 75.0 21.7	12.8 95.0 22.0	20.0 20.0 20.0	12.2 70.0 20.7	9.47 90.0 17.1
Average	43.8 68.4 **52.4**	37.7 37.4 28.0	39.9 58.9 42.9	34.3 80.2 43.8	48.4 29.1 34.9	48.9 39.5 32.8	31.1 58.5 33.8

TABLE III
PRECISION, RECALL AND F1 SCORES OF CROSS-PROJECT DEFECT PREDICTION. ALL THE SCORES ARE MEASURED BY PERCENTAGE. THE BEST F1 SCORES ARE HIGHLIGHTED IN BOLD.

Source	Target	Cross-project		Within-project
		DDA	Embedding	
		P R F1	P R F1	P R F1
Nuvolabase	Checkstyle	79.0 57.6 **66.7**	54.5 38.8 45.3	74.6 84.7 79.3
OrientDB	Checkstyle	94.3 29.4 44.8	54.7 48.2 **51.3**	
Checkstyle	Nuvolabase	45.5 36.4 **40.4**	27.0 52.2 35.6	33.8 66.0 44.7
Traccar	Nuvolabase	44.2 36.4 **40.0**	27.1 41.5 32.8	
Nuvolabase	OrientDB	57.1 16.7 **25.8**	16.2 31.9 21.5	32.9 47.9 39.0
Traccar	OrientDB	25.4 43.1 **31.9**	18.5 45.1 26.3	
Nuvolabase	Traccar	13.9 85.0 **23.9**	16.7 15.0 15.8	33.9 75.0 46.7
Checkstyle	Traccar	16.0 20.0 **17.8**	9.80 50.0 16.4	
Average		46.9 40.6 **36.4**	28.1 40.3 30.6	43.8 68.4 52.4

TABLE IV
TRAINING TIME OF THE PROPOSED DDA APPROACH

Project	Time (s)
Checkstyle	10.2
Nuvolabase	62.5
OrientDB	59.2
Traccar	5.67
Average	34.4

D. Results

This section presents our experimental results. We examine the performance of our proposed DDA approach in both within-project and cross-project defect prediction setting. In the within-project setting, we use the source code of an older version of a project to construct the DDA model and evaluate the model based on the source code of the newer version of the project. In the cross-project setting, we randomly pick one project as a source project to build the DDA model and use the model to predict defects for a target project that is randomly picked from a set of projects that excludes the source project.

We answer the following research questions:

RQ1: In within-project defect prediction, does our proposed approach outperform baselines?

Table II shows the precision, recall and F1 score of different defect prediction models. The highest F1 scores are highlighted in bold. For example, the F1 score of our approach is 46.7% for Traccar project, while the best F1 score is only 23.9% for approaches that use embedding features (using decision tree), and the best F1 score is 20.7% for approaches that use AST features (using logistic regression). On average, the best baseline that uses AST features achieves an F1 score of 34.9%, while the best baseline that uses embedding features constructed following Wang et al. [40] approach achieves an F1 score of 43.8%. Our DDA approach beats these two baselines by achieving an F1 score of 52.4%. The results demonstrate that we can improve the F1 score by 19.63% when compared with the best baseline.

RQ2: In cross-project defect prediction, does our proposed approach outperform baselines?

We evaluate eight pairs of projects. For each pair, we take two different projects for training and testing. Table III presents the precision, recall and F1 scores of our proposed method (DDA) vs. best defect prediction models constructed using embedding features. We employ naïve Bayes algorithm to build defect prediction model from embedding features since this algorithm achieves the best F1 score in the within-project setting (see Table II). The best F1 scores are highlighted in bold. For example, when the source project is Nuvolabase (training) and the target project is Checkstyle (testing), our DDA achieves an F1 score of 66.7% whereas the best defect prediction model using embedding features only achieves an F1 score 45.3%. In average, DDA achieves an F1 score of 36.4%, which improves by 18.95% in term of F1 score compared to the best model that uses embedding features.

RQ3: What is the training time of proposed approach?

We run experiments on a NVIDIA DGX-1 [2] machine to construct the DDA model. We keep track of the training time that our server needs to build the DDA model on the four software projects in the within-project setting. Table IV shows the training time to build the DDA model. In average, the training time for our proposed approach varies from 5.67 seconds (Traccar) to 62.5 seconds (Checkstyle). On average, it takes 34.4 seconds to build the DDA model. It shows that our DDA is applicable in practice.

E. Threats to validity

¡¡¡¡¡¡¡ HEAD Threats to validity includes threats to internal, external, and construct validity. To minimize threats to internal validity, we have made sure that our implementations are correct. For baseline, Wang et al. [40] are unable to share their source code since their approach is under US patent application. Thus, we reimplement their approach by following the description in their paper and querying with the first author[3]. Regarding threats to external validity, our dataset consists only of four open source Java projects. However, the projects has varying statistics in average buggy rates and number of source code files. In the future, we will minimize threats to external validity further by experimenting on more projects with more varying statistics and also projects that are closed source and written in different programming languages (i.e., C++, Python, etc.). To minimize threats to construct validity, we use of evaluation metrics that are common in defect prediction [23], [24], [29]. ======= Threats to validity includes threats to internal, external, and construct validity. To minimize threats to internal validity, we have made sure that our implementations are correct. For baseline, Wang et al. [40] are unable to share their source code since their approach is under US patent application. Thus, we reimplement their approach by following the description in their paper and querying with the first author[4]. Regarding threats to external validity, our dataset consists only of four open source Java projects. However, the projects has varying statistics in average buggy rates and number of source code files. In the future, we will minimize threats to external validity further by experimenting on more projects with more varying statistics and also projects that are closed source and written in different programming languages (i.e., C++, Python, etc.). To minimize threats to construct validity, we use of evaluation metrics that are common in defect prediction [23], [24], [29]. ¿¿¿¿¿¿¿ ced81307fd2a1b168a7c797437cfcc3f71218ea0

IV. Related Work

A. Defect Prediction

The software defect prediction problem has been studied in the past decade [29], [24], [23], [44], [12], [28], [30], [38]. Traditional approaches in defect prediction often manually extract features from historical defect data to construct machine learning classification model [24]. Moser et al. [26] employed the number of revisions of a file, age of a file, number of authors that checked a file, etc. for defect prediction. Lee et al. [19] introduced 56 novel micro interaction metrics (MIMs) leveraging developers' interaction information stored in the Mylyn data, and showed that MIMs significantly improve the performance of defect classification. Jiang [12] showed that individual characteristics and collaboration between developers were useful for defect prediction.

Based on these features, classification models are built to predict the defect among program elements. Elish et al. [6] estimated the capability of Support Vector Machine (SVM) [35] in predicting defect-prone software modules and showed that the prediction performance of SVM is generally better than eight statistical and machine learning models in NASA datasets. Amasaki et al. [1] employed Bayesian belief network (BBN) [21] to predict the amount of residual faults of a software product. Khoshgoftaar et al. [15] showed that tree-based machine learning algorithms are effective for defect prediction. Jing et al. [13] proposed to use dictionary learning technique to predict software defect. They introduced a cost-sensitive discriminative dictionary learning (CDDL) approach for software defect classification and prediction.

Wang et al. [40] employed DBN to learn semantic features from AST. The features are used to build a defect prediction model. They are learned from source files without considering their defect label and thus are suboptimal for defect prediction task. To overcome this problem, we propose a DDA model that acts as an end-to-end learning framework to build semantic features and defect prediction model in one stage. On average, in terms of F1 score, our approach outperforms Wang et al. by a substantial margin.

B. Deep Learning in Software Engineering

Deep learning algorithms have been widely used to improve research tasks in software engineering in the past few years. Lam et al. [17] combined deep neural network (DNN) [9] with rVSM [43], a revised vector space model, to improve the performance of bug localization. Raychev et al. [32] reduced the problem of a code completion to a natural-language processing problem of predicting sentences' probabilities. They used recurrent neural network [25] to predict the probabilities of subsequent words in a sentence. Mou et al. [27] proposed a tree-based convolutional neural network (TBCNN) for programming language processing. Results of their experiment showed that the effectiveness of TBCNN in two different software engineering tasks: classifying programs according to functionality, and detecting code snippets of certain patterns. Pascanu et al. [31] employed recurrent neural network to build a malware classification model in software system. Yuan et al. [42] adopted DBN to predict mobile malware in Android platform. Their experimental results showed that a deep learning technique is suitable for predicting malware in software system.

[2] https://www.nvidia.com/en-us/data-center/dgx-1/

[3] We provide the source code of our implementation at https://drive.google.com/open?id=0B3FfOb7cKbK3UWlRMEhUX3FJU2s

[4] We provide the source code of our implementation at https://drive.google.com/drive/folders/0B3FfOb7cKbK3UWlRMEhUX3FJU2s

V. Conclusion and Future Work

This paper presents a new deep discriminative autoencoder (DDA) approach to achieve an effective software defect prediction. DDA provides an end-to-end learning approach to simultaneously learn embedding features that can well represent token vectors extracted from programs' ASTs, and build an accurate classification model for defect prediction. Empirical studies on four software projects show that our approach significantly outperforms the existing defect prediction approaches. Specifically, our approach improve the F1 score by 19.63% and 18.95% when compared with the state-of-the-art approach for both within-project and cross-project setting, respectively.

While DDA offers a powerful approach for defect prediction, there remains room for improvement. In the future, we plan to improve the effectiveness of our approach, potentially by either generating better features or enhancing our DDA model. We also plan to experiment on more dataset and evaluate our approach more thoroughly.

References

[1] S. Amasaki, Y. Takagi, O. Mizuno, and T. Kikuno, "A bayesian belief network for assessing the likelihood of fault content," in *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*. IEEE, 2003, pp. 215–226.

[2] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*. ACM, 2008, p. 23.

[3] C. M. Bishop, "Pattern recognition," *Machine Learning*, vol. 128, pp. 1–58, 2006.

[4] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[5] F. B. e Abreu and R. Carapuça, "Candidate metrics for object-oriented software within a taxonomy framework," *Journal of Systems and Software*, vol. 26, no. 1, pp. 87–96, 1994.

[6] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *Journal of Systems and Software*, vol. 81, no. 5, pp. 649–660, 2008.

[7] R. Harrison, S. J. Counsell, and R. V. Nithi, "An evaluation of the mood set of object-oriented software metrics," *IEEE Transactions on Software Engineering*, vol. 24, no. 6, pp. 491–496, 1998.

[8] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 78–88.

[9] R. Hecht-Nielsen *et al.*, "Theory of the backpropagation neural network." *Neural Networks*, vol. 1, no. Supplement-1, pp. 445–448, 1988.

[10] G. E. Hinton, "Deep belief networks," *Scholarpedia*, vol. 4, no. 5, p. 5947, 2009.

[11] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant, *Applied logistic regression*. John Wiley & Sons, 2013, vol. 398.

[12] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 279–289.

[13] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu, "Dictionary learning based software defect prediction," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 414–423.

[14] T. M. Khoshgoftaar, K. Gao, and N. Seliya, "Attribute selection and imbalanced data: Problems in software defect prediction," in *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, vol. 1. IEEE, 2010, pp. 137–144.

[15] T. M. Khoshgoftaar and N. Seliya, "Tree-based software quality estimation models for fault prediction," in *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*. IEEE, 2002, pp. 203–214.

[16] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[17] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (n)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 476–481.

[18] T.-D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: Better together," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 579–590.

[19] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, "Micro interaction metrics for defect prediction," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 311–321.

[20] C. D. Manning, P. Raghavan, H. Schütze *et al.*, *Introduction to information retrieval*. Cambridge university press Cambridge, 2008, vol. 1, no. 1.

[21] K. T. McAbee, N. P. Nibbelink, T. D. Johnson, and H. T. Mattingly, "Bayesian-belief network model."

[22] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[23] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE transactions on software engineering*, vol. 33, no. 1, 2007.

[24] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Automated Software Engineering*, vol. 17, no. 4, pp. 375–407, 2010.

[25] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, "Recurrent neural network based language model." in *Interspeech*, vol. 2, 2010, p. 3.

[26] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 181–190.

[27] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang, "Tbcnn: A tree-based convolutional neural network for programming language processing," *CoRR, abs/1409.5718*, 2014.

[28] N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*. IEEE, 2007, pp. 364–373.

[29] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 382–391.

[30] T. T. Nguyen, T. N. Nguyen, and T. M. Phuong, "Topic-based defect prediction (nier track)," in *Proceedings of the 33rd international conference on software engineering*. ACM, 2011, pp. 932–935.

[31] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas, "Malware classification with recurrent networks," in *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*. IEEE, 2015, pp. 1916–1920.

[32] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 419–428.

[33] L. Rosasco, E. De Vito, A. Caponnetto, M. Piana, and A. Verri, "Are loss functions all the same?" *Neural Computation*, vol. 16, no. 5, pp. 1063–1076, 2004.

[34] S. R. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE transactions on systems, man, and cybernetics*, vol. 21, no. 3, pp. 660–674, 1991.

[35] J. A. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural processing letters*, vol. 9, no. 3, pp. 293–300, 1999.

[36] V. N. Vapnik and V. Vapnik, *Statistical learning theory*. Wiley New York, 1998, vol. 1.

[37] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *Journal of Machine Learning Research*, vol. 11, pp. 3371–3408, 2010.

[38] J. Wang, B. Shen, and Y. Chen, "Compressed c4. 5 models for software defect prediction," in *Quality Software (QSIC), 2012 12th International Conference on*. IEEE, 2012, pp. 13–16.

[39] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 434–443, 2013.

[40] S. Wang, T. Liu, and L. Tan, “Automatically learning semantic features for defect prediction,” in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 297–308.

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

[42] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, “Droid-sec: deep learning in android malware detection,” in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 371–372.

[43] J. Zhou, H. Zhang, and D. Lo, “Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports,” in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 14–24.

[44] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting defects for eclipse,” in *Proceedings of the third international workshop on predictor models in software engineering*. IEEE Computer Society, 2007, p. 9.