Deep Discriminative Autoencoder for Software 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. Extensive 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 8.6% in terms of F1 score for the WP problems. For the CP problems, DDA outperforms other methods by 5.8% in terms of F1.**

I. Introduction

Software defect prediction techniques [9], [13], [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 or any bugs. 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. Specifically, the 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 [13], [6], [22], [5], [8]. Meanwhile, machine learning algorithms have been used to construct a defect prediction model, such as decision tree, logistic regression, and naïve Bayes, etc [14]. However, traditional approaches often fail to capture the semantic differences of programs since they cannot learn code structure of different semantics. [what do we mean by "semantic" here? And why traditional ML cannot capture semantic differences?]

Wang et al. [40] recently developed a deep belief network (DBN) model [11] to automatically learn embedding features that are a compressed representation of token vectors extracted from a program's abstract syntax tree (AST). The learned features are then utilized as the 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 identifier 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. Our proposed framework extends the deep autoencoder model [31] by augmenting additional hidden layers that map the embedding features to an output layer where the defect classification is made. We summarize the key contributions of this paper below:

- We develop a powerful 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 extensive 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 SectionIV. Review of key related works is given in Section V. Finally, Section VI concludes this paper.

II. Proposed Approach

A. Defect Prediction

In this work, we focus on file-level defect prediction task, which consists of two main steps. The first step is to label source code files as buggy or clean and then extract traditional features of these files [what do we mean by "traditional features"?]. These traditional features are introduced in [40], [22], [4]. The second step is to construct a defect prediction

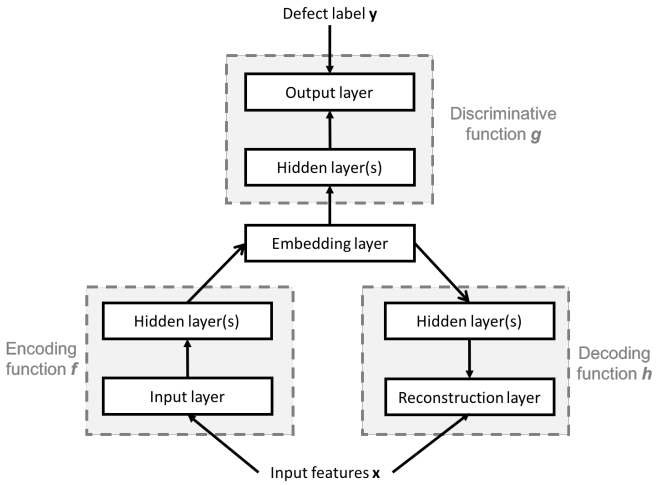

Fig. 1. Our proposed DDA model architecture

model [3] to predict whether a new source code file is buggy or clean. We refer to the software version used for building our defect prediction model as training data and the one used to evaluate the built model as testing data. [Is this last sentence really necessary? What does "software version" mean specifically in this context?]

B. Parsing Source Code and Generating Input Features

Following Wang et al.'s approach [40], we extract syntactic information from source code based on Java Abstract Syntax Tree (AST) [30]. For each Java source code file, we extract a sequence of AST node tokens of the following types: 1) nodes of method invocations and class instance creations, 2) declaration nodes, i.e., method declarations, type declarations, and enum declarations, and 3) control-flow nodes such as while and for statements, if statements, catch clauses, etc.

However, in contrast to Wang et al.'s approach which encodes the extracted AST tokens as unique integers and treats them equally, we encode the tokens using a term frequency–inverse document frequency (TF-IDF) [20]. The TF-IDF features are then used as "raw" inputs to our DDA model. In this case, the TF-IDF representation would be better than simple integer encoding in capturing the relative importance of a token within a particular source code file.

C. 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 $\mathcal{X} = \{x_1, \ldots, x_i, \ldots, x_n\}$ denote 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 the two tasks in one shot for tackling defect prediction problem. Specifically, DDA jointly learns three (non-linear) functions, namely: (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) an decoding function h that reconstructs the input features from the embedding representation. Fig. 1 presents the architecture of our DDA model realizing the three functions. Each function may be represented using one or more hidden layers of neuron.

These two functions f and f' can be learned by minimizing the following objective function:

$$\min_{f,f'} \sum_i \mathcal{L}(f(x_i), x_i) + \theta \mathcal{L}'(f'(x_i), y_i) + \lambda \Omega(f, f') \quad (1)$$

where $\mathcal{L}(\cdot,\cdot)$ and $\mathcal{L}'(\cdot,\cdot)$ are the empirical loss of the semantic features and the defect prediction functions, respectively. θ is the predefined value for weighting the two loss functions. $\Omega(f, f')$ is the regularization term imposed on the two functions. The trade-off between the empirical loss and the regularization term is controlled by λ.

The overall framework of DDA is shown in Figure 1. The DDA model contains three different layers: input layer, hidden layer, and output layer. Given a source code file, the features extracted in Section II-B are fed to the input layer while the corresponding defect label is fed to the output layer. The network consisting of input layer, hidden layer, and input layer represents an encoder-decoder model. The encoder-decoder model is required to learn semantic features. Note that our encoder-decoder model is inspired by autoencoder [31], which is an unsupervised learning technique. The original autoencoder only learns the function $f : \mathcal{X} \longmapsto \mathcal{X}$ so that the output values $\hat{\mathcal{X}}$ are similar to input values $\mathcal{X}$. On the other hand, DDA attempts to learn semantic features and optimize defect prediction task by taking into account two functions, i.e., f and f', which represents the semantic features and defect prediction function, respectively. f' is learned through the connection between the hidden layer and the output layer. According to Figure 1, our model optimizes two loss functions, i.e., $\mathcal{L}$ and $\mathcal{L}'$ to construct the defect prediction model. In encoder-decoder model, we employ a fully connected neural network for learning to convert low level features from source code files to semantic features. At the same time, our network learns to determine whether the given source code file is buggy based on the semantic features.

D. Imbalanced Problem in Defect Prediction

In defect prediction tasks, often times there are only a few source code files that contain bugs while the other source code files are *clean* [15]. This consequently makes the labeled data to be imbalanced. This imbalanced nature increases the learning difficulty. For this reason, imbalanced class learning, which specializes in tackling classification problems involving imbalanced data, is helpful for defect prediction problem [39]. To address this imbalanced data issue, we propose a balanced random sampling procedure when picking a data instance to

update our DSSL network weights. In particular, we select a random instance from each the positive and negative instance pools to mitigate the issue of skewed distribution. This mitigates the issue of imbalanced data in defect prediction.

E. Setting for Training Deep Semi-supervised Learning

In our setting, the number of hidden layers, the number of nodes in each hidden layer, the number of iterations, and θ are chosen by performing cross validation on training data. By default, the number of hidden layers, the number of nodes in each hidden layer, and number of epochs are selected as 2, 1000-100, and 75, respectively. We employ Adam optimization [17], which is popular optimization method in deep learning community, to optimize the two loss functions for constructing DDA.

III. Experimental Results

We conduct several experiments to study the performance of the proposed approach and compare it with existing traditional approaches.

A. Evaluation Metrics

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

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{4}$$

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 can reveal more defects given a project. F1 takes a consideration of both precision and recall.

B. Datasets

We perform several steps to create our benchmark dataset. Firstly, we fetch 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 each remaining 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 commit 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 for the training version.

We then randomly select four projects as the dataset for our preliminary experiment. Table I shows statistics on this dataset. In average, our dataset contains around 783.875 source files with bug rate of 17.4%, showing the imbalanced problem in defect prediction [39], [15].

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 [3] to build defect prediction models for each traditional features. These algorithms are widely used in software engineering [40], [39], [14] 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 [35].
- Logistic regression is a well-known classification model that employs in various application such as: health, statistics, data analysis, etc. [12].
- Naive Bayes classifier, which are highly scalable, is a simple probabilistic classifiers based on applying Bayes' theorem [37].

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 this 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 traditional models generated

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
COMPARISON BETWEEN DDA AND THREE CLASSIFICATION ALGORITHMS (DECISION TREE, LOGISTIC REGRESSION, AND NAIVE BAYES) USING SEMANTIC FEATURES AND AST FEATURES. P, R, AND F1 DENOTE PRECISION, RECALL AND F1 SCORE RESPECTIVELY AND ARE MEASURED BY PERCENTAGE. THE BEST F1 SCORES ARE HIGHLIGHTED IN BOLD.

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

by different machine learning algorithms using semantic features?

We run experiments on four popular software projects. For each project, we construct defect prediction models using the training version and evaluate them in the testing version. Note that the training and testing are collected in two different timelines (i.e., version as of January 1^{st}, 2015 and version as of July 1^{st}, 2015). 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 of our approach is 46.7% in Traccar project, while the best F1 is only 23.9% with embedding features (using decision tree), and the best F1 is 20.7% with AST features (using logistic regression). On average, the best baseline that uses AST features achieve an F1 score of 34.9%, while the best baseline that uses semantic features constructed following [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 to be 8.6% higher when compared with the best baseline.

RQ2: In cross-project defect prediction, does our proposed approach outperform traditional models generated by different machine learning algorithms using semantic features?

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. the best defect prediction models constructed based on embedding features. We employ naive Bayes algorithm to build defect prediction model from embedding features since this algorithm shows the best F1 in within-project (see Table II). The best F1 scores are also in bold. For example, when the source project is Nuvolabase (training) and the target project is Checkstyle (testing). Our DDA achieves a F1 score of 66.7% whereas the best defect prediction model used embedding features only achieves 45.3%. In average, DDA achieves a F1 score of 36.4%, which is a 5.6% higher than F1 score of the best model that uses semantic features. It shows that our proposed DDA improves the performance of cross-project defect prediction.

RQ3: What is the training time of proposed approach? We run experiments on a NVIDIA DGX-1 [1] 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 for within-project problem. Table IV shows the training time which needs to build DDA model. In average, the training time for our proposed method varies from 5.67

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

seconds (Traccar) to 62.5 seconds (Checkstyle). On average, it takes 34.4 seconds to build the DDA model. Hence it proves that our DDA is applicable in practice.

IV. Threats to validity

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. 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].

V. Related Work

A. Defect Prediction

The software defect prediction has been studied in the past decade [29], [24], [23], [44], [13], [28], [32], [38]. However, the 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. Nagappan et al. [28] extracted features by considering relationship between its software dependencies, churn measures and post-release failures to build a classification model for defect prediction. Lee et al. [19] introduced 56 novel micro interaction metrics (MIMs) leveraging developers' interaction information stored in the Mylyn data, and shown that MIMs significantly improve the performance of defect classification. Jiang [13] 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. [7] estimated the capability of Support Vector Machine (SVM) [36] 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. [16] showed that the Tree-based machine learning algorithms are efficiently in defect detection. Jing et al. [14] proposed to use the dictionary learning technique to predict software defect. Typically, they introduced a cost-sensitive discriminative dictionary learning (CDDL) approach for software defect classification and prediction.

The main differences between our approach and traditional approaches are as follows. First, existing approaches to defect prediction are based on manually encoded traditional features which are not sensitive to programs' semantic information, while our approach automatically learns semantic features using semi-supervised autoencoder. Second, these features are automatically employed to construct classification model for defect prediction tasks.

B. Deep Learning in Software Engineering

Recently, deep learning algorithms have been widely used to improve research tasks in software engineering. Lam et al. [18] combined deep neural network (DNN) [10] with rVSM [43], a revised vector space model, to improve the performance of bug localization. Raychev et al. [34] reduced the problem of code completion to a natural-language processing problem of predicting probabilities of sentences and used recurrent neural network [25] to predict the probabilities of the next token. Mou et al. [27] proposed tree-based convolutional neural network (TBCNN) for programming language processing. The results showed that the effectiveness of TBCNN in two different program analysis tasks: classifying programs according to functionality, and detecting code snippets of certain patterns. Pascanu et al. [33] employed recurrent neural network to build malware classification model in software system. Yuan et al. [42] adopted deep belief network (DBN) [11] to predict mobile malware in Android platform. The experimental results showed that deep learning technique is especially suitable for predicting malware in software system.

Yang et al. [41] leveraged DBN to generate features from existing features and used these new features to predict whether a program element contains bugs. It showed that the deep learning algorithm helps to discover more bug than tradition approaches on average across from six large software projects. The existing features were manually designed based on change level: i.e., the number of modified subsystems, code added, code deleted, the number of files change, etc. In 2016, Wang et al. [40] also employed DBN to learn semantic features from source code. However, the existing features were extracted from abstract syntax tree since [38] claimed that Yang features [41] were fail to distinguish the semantic difference among source code. The evaluation on ten popular source projects showed that the semantic features significantly improved the performance of defect detection. Different to the existing works that semantic features and defect prediction model are built independently, thus the semantic features only learn from source code without considering the label of this program element which may decrease the performance of defect prediction model. To tackle this problem, we propose a deep discriminative autoencoder to build classification model for solving defect prediction problem. We evaluate the effectiveness of our proposed approaches against Wang approaches [38] and the traditional machine learning algorithms (i.e., naive bayes, logistic regression, and random forest) on four popular software projects .

VI. 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 simulateneously learn embedding features that can well represent token vectors extracted from programs' ASTs, and to 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 improves on average by 8.6% and 5.8% for the within-project and cross-projects problems in terms of F1 score, respectively.

While DDA offers a powerful approach for defect prediction, there remains room for improvement. For example, DDA currently takes as its input the TF-IDF features constructed from AST nodes, thus ignoring the meaning of AST nodes within source codes of a program. In the future, we wish to learn the semantics of AST nodes by employing word embedding representation. We also plan to develop a more sophisticated deep learning method to better capture sequence information within a defect prediction task.

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. Chakradeo, B. Reaves, P. Traynor, and W. Enck, "Mast: Triage for market-scale mobile malware analysis," in *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*. ACM, 2013, pp. 13–24.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[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] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[31] A. Ng, "Sparse autoencoder," *CS294A Lecture notes*, vol. 72, no. 2011, pp. 1–19, 2011.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

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

[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.