Deep Semi-Supervised 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 using a classification algorithm. However, the feature construction stage is usually done based on source files in an unsupervised manner (i.e., without considering known defect labels), which may limit the effectiveness of learned features. In light of this deficiency, we propose in this paper a new deep semi-supervised learning approach that performs end-to-end training to simultaneously construct discriminative features as well as accurate classification model for a more effective defect identification. Extensive experimental results on four popular software projects show that our approach significantly outperforms traditional approaches on both within-project (WP) and cross-project (CP) defect prediction. Typically, our deep semi-supervised learning improves WP on average by 8.6% in F1. For CP, our approach outperforms the defect prediction models using traditional features by 5.8% in F1.**

I. Introduction

Software defect prediction techniques [10], [14], [46] have been proposed to detect defects among program elements to help developers to reduce their testing efforts, thus leading to reduce software development costs. Defect prediction tries to construct defect prediction models from software history data, and uses these models to predict whether new instances of code regions, e.g., files, changes, and methods, contain defects or any bugs. Traditional approaches try to construct accurate defect prediction models following two different directions: the first direction focuses on manually designing a set of features so that it can represent defects more effectively; the second direction aims to build a new machine learning algorithm to improve the prediction models.

In the past, most researchers have manually designed features to filter buggy source files from non-buggy files. McCabe et al. [23] features focus on a complexity measure for the program elements, CK features [5] based on function and inheritance counts to understand the development of software projects, whereas MOOD features [9] tried to provide an overall assessment of a software system. The other features are constructed based on source code changes like, the number of lines of code added, removed, etc. [14], [6]. On the other hand, many machine learning algorithm have been widely used for software defect prediction, including decision tree, logistic regression, Naive Bayes, etc [15]. However, traditional approaches fail to distinguish code regions of different semantics.

To bridge the gap between programs' semantic information and features used for defect prediction, Wang et al. [42] employed Deep Belief Network (DBN) [12] to automatically learn features from token vectors extracted from programs' ASTs, and then utilize these features to train a defect prediction model. However, Wang approaches [42] build semantic features and defect prediction model independently. Typically, semantic features only learn from source files without considering the true label of this program element. Moreover, token values are mapped to unique integer identifier without reflecting how important a token in program element files's. Hence the semantic features may fail to optimize the defect prediction model.

To tackle this problem, we propose a deep semi-supervised learning (DSSL) approach allowing to extract semantic features and optimize the prediction model in one stage. Our proposed framework takes advantage of antoencoder [32] to construct DSSL model for defect prediction.

This paper makes the following contributions:

- We propose to leverage a powerful representation learning algorithm, namely deep semi-supervised learning, to construct the defect prediction model.
- Our evaluation results on four popular software Java projects shows that our approach significantly improve the performance of defect prediction by 8.6% and 5.4% on both within-project and cross project defect prediction respectively in term of F1 compared to traditional approaches.

The rest of this paper is summarized as follows. Section II briefly presents the defect prediction problem and our deep semi-supervised learning. Section III shows the experimental results of our approaches. Section IV presents threat to validity. Section V and Section VI describe the related work and conclusion of our paper.

II. Proposed Approach

A. Defect Prediction

The overall process of our file-level defect prediction follows two specific steps. The first step is to label source code files as buggy or clean and then extracts traditional features of these files. These traditional features are introduced in [42], [23], [4]. The second step is to construct a defect prediction 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.

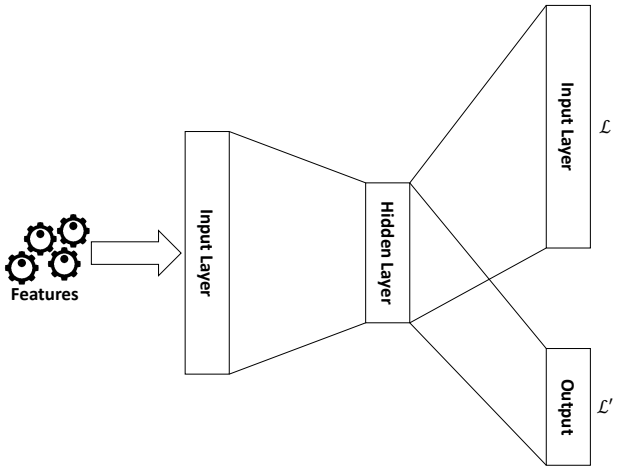

Fig. 1. Deep Semi-supervised Learning Framework

B. Parsing Source Code and Generating Features

In our approach, we follow Wang et al. [42] approach to extract source code information. Typically, the syntactic information from source code is collected based on Java Abstract Syntax Tree (AST) [31]. For each Java source code file, we extract a sequence of AST node tokens of these 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 statements, catch clauses, if statements, for statements, etc. Unlike Wang approach which encode the extracted tokens as unique integers and use them as features, we encode the tokens using a term frequency - inverse document frequency (TF-IDF) [21] and consider them as features for our framework (i.e., deep semi-supervised learning). These features reflects how important a token is in its corresponding source code file.

C. Deep Semi-supervised Learning

The goal of defect prediction is to detect source code files that may contain bug in the future.

Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ denotes the set of source code files in a software project and $\mathcal{Y} = \{y_1, y_2, \ldots, y_n\}$ represents the set of labels for the source code files, where n is the number of source code files in the project. Note that the source code file is labeled as 1 if it contains bug, otherwise it will be labeled as 0 which means that it is clean from bug. Unlike traditional approaches [43], [42] that independently learn semantic features and construct defect prediction model, our deep semi-supervised learning (DSSL) combines the two tasks for tackling the defect prediction problem. Typically, we attempt to learn a semantic features function $f : \mathcal{X} \longmapsto \mathcal{X}$ and a defect prediction function $f' : \mathcal{X} \longmapsto \mathcal{Y}$, $y_i \in \mathcal{Y} = \{0, 1\}$ indicates whether a source code file $x_i \in \mathcal{X}$ contains a bug. These two functions f and f' can be learned by minimizing the following objective function:

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

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 DSSL is shown in Figure 1. The DSSL 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 are inspired by autoencoder [32], which is an unsupervised learning technique. The original autoencoder only learn 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, DSSL attempts to learn semantic features and optimize defect prediction task, thus it takes 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 on 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* [16]. 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 [41]. 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 iteration are selected as 2, 1000-100, and 75, respectively. We employ Adam optimization [18], which is popular optimization method in deep learning community, to optimize the two loss functions for constructing DSSL.

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 results, we employ three different metrics: *Precision*, *Recall* and *F1*. These evaluation metrics are widely used to evaluate the performance of defect prediction [24], [25], [30] as well as information retrieval binary classification [21]. Typically, precision is the fraction of retrieved instances that are relevant, recall is the fraction of relevant instances that are retrieved, whereas F1 combines both precision and recall to measure the performance of our model. Below is the equation 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 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 4 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 17.4175, showing the imbalanced problem in defect prediction [41], [16].

C. Baselines

To evaluate the performance of our approach in defect prediction, we compare with the traditional defect prediction models constructed based on state-of-the-art semantic features following Wang et al. approaches [42]. Typically, they tried to employ Deep Belief Network [12] to automatically learn semantic features from token vectors extracted from programs' AST. They also proved that their semantic features significantly improve the performance of defect prediction in software engineering domain.

We employ three popular machine learning algorithms [3] which are widely used in software engineering domain [42], [41], [15] to build defect prediction models. They are described as following:

- Decision tree is used to build a predictive model about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves). This algorithm is very popular in statistics, data mining and machine learning [37]. Tree models where the target variable can take a finite set of values are called classification trees; in these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels.
- Logistic regression is a predictive analysis and used to describe data and to explain the relationship between one dependent binary variable and one or more nominal, ordinal, interval or ratio-level independent variables. Logistic regression is used in various applications like: health, statistics, data analysis, etc. [13].
- Naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem [39] with strong (naive) independence assumptions between the features. Naive Bayes classifiers are highly scalable, requiring a number of parameters linear in the number of variables (features/predictors) in a learning problem. For some types of probability models, Naive Bayes classifiers can be trained very efficiently in a supervised learning setting. In many practical applications, parameter estimation for naive Bayes models uses the method of maximum likelihood [34].

D. Results

This section presents our experimental results. We examine the performance of our propose approaches, i.e., deep semi-supervised learning (DSSL) in both within-project and cross-

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 DEEP SEMI-SUPERVISED LEARNING AND TWO BASELINES FEATURES (SEMANTIC FEATURES AND AST FEATURES) USING THREE DIFFERENT CLASSIFICATION ALGORITHMS (DECISION TREE, LOGISTIC REGRESSION, AND NAIVE BAYES). 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	DSSL	Semantic			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
		DSSL	Semantic	
		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
TIME COST USED TO CONSTRUCT DEEP SEMI-SUPERVISED LEARNING MODEL.

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

project defect prediction. In the within-project, we use the source code of an older time to construct DSSL model and evaluate this model based on source code of a recent time. For the cross-project, we randomly pick 1 project as source project to build defect prediction model and use the model to predict defect for a new project called target project.

We are trying to answer the following research questions:

RQ1: Does our proposed framework outperform the traditional models generated by different machine learning algorithms based on semantic features in within-project defect prediction?

We run the experiments on four popular of software project, each of which uses two versions of the same project collected in two different time-lines (see Table I). The training data, which is the older version of these projects, are used to construct defect prediction models, and the testing data is used to evaluate the performance of our prediction models. Table II shows the precision, recall and F1 of the defect prediction experiments. On average, AST features achieve best F1 of 34.9%, the semantic features constructed following [42] approach achieves the best F1 of 43.8% using naive Bayes classification algorithm, and our deep semi-supervised learning (DSSL) achieves a F1 of 52.4%. The results demonstrate that we can improve the defect prediction F1 by 8.6% on average of four software projects.

RQ2: Does our proposed framework outperform the traditional models generated by different machine learning algorithms based on semantic features in cross-project defect prediction?

We also compare our DSSL against Wang et al. [42] approach. Note that we also provide a bench-mark of within-project for a fair comparison. We evaluate eight pairs of cross-project defect prediction experiments. For each experiment, we take two different projects for training set and testing set. Table III presents the precision, recall and F1 scores of our proposed method (DSSL) vs. the best defect prediction models constructed based on semantic features. In average, DSSL achieves 36.4% which is 5.6% higher than the 30.6% of semantic features.

RQ3: What is the time cost of proposed framework? We run experiments on a NVIDIA DGX-1 [1] machine to construct deep semi-supervised learning model (DSSL). We keep track the time cost that our server needs to build the model on four software projects. Table IV shows the time cost for constructing DSSL. The time cost for our proposed method vary from 5.67 seconds (tananaev) to 62.5 seconds

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

(Checkstyle). On average, it takes 34.4 seconds to build DSSL model.

IV. Threats to validity

The projects for this paper contain a large variance in average buggy rates and program elements. Typically, we choose the projects which have more than 200 program elements and more than 10 bugs for running the experiments. However, there is a chance that our projects are not generalizable enough to represent all software projects. Thus, the proposed approach might get better or worse results for the other projects. Furthermore, the proposed semi-supervised autoencoder is only evaluated on open source Java projects. In the future work, we would like to employ the proposed approach on close source software and projects written in different languages (i.e., C++, Python, etc.).

V. Related Work

A. Defect Prediction

The software defect prediction has been studied in the past decade [30], [25], [24], [46], [14], [29], [33], [40]. However, the traditional approaches in defect prediction often manually extract features from historical defect data to construct machine learning classification model [25]. McCabe et al. [23] introduced a graph-theoretic complexity measure for the control program elements which can be considered as a feature in defect prediction. CK features [5] focused on understanding of software development process, while MOOD features [9] provided an overall assessment of a software system to manage the software development projects. These features are widely used in defect prediction. Moser et al. [27] employed the number of revisions of a file, age of a file, number of authors that checked a file, etc. to defect prediction. Nagappan et al. [29] extracted features by considering relationship between its software dependencies, churn measures and post-release failures to build classification model for defect prediction. Lee et al. [20] 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 [14] 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) [38] 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) [22] to predict the amount of residual faults of a software product. Khoshgoftaar et al. [17] showed that the Tree-based machine learning algorithms are efficiently in defect detection. Jing et al. [15] 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. [19] combined deep neural network (DNN) [11] with rVSM [45], a revised vector space model, to improve the performance of bug localization. Raychev et al. [36] o reduced the problem of code completion to a natural-language processing problem of predicting probabilities of sentences and used recurrent neural network [26] to predict the probabilities of the next token. Mou et al. [28] 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. [35] employed recurrent neural network to build malware classification model in software system. Yuan et al. [44] adopted deep belief network (DBN) [12] 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. [43] 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. [42] also employed DBN to learn semantic features from source code. However, the existing features were extracted from abstract syntax tree since [40] claimed that Yang features [43] 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 semi-supervised learning to build classification model for solving defect prediction problem. We evaluate the effectiveness of our proposed approaches against Wang approaches [40] and the traditional machine learning algorithms (i.e., naive

bayes, logistic regression, and random forest) on four popular software projects .

VI. Conclusion

Our paper presents a deep semi-supervised learning to optimize defect prediction model. Typically, we take advantage of deep learning autoencoder to learn semantic features from token vectors extracted from programs' ASTs automatically, and optimize these feature to construct classification model for predicting defects. Our evaluation on four software projects shows that our approaches could significantly improve the performance of defect prediction compared to two traditional approaches, i.e., AST features and semantic features. In the future, we would like to extend our automatically semantic feature generation approach to C/C++ projects for defect prediction. In addition, it would be promising to leverage our approach to automatically build defect prediction model at different levels, i.e., change level, module level, or package level, etc. instead of file level.

VII. Acknowledgments

The authors thank the anonymous researchers for their feedback which help to improve the paper. This work has been partially supported by the National Research Funding of Singapore.

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] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. IEEE, 2003, pp. 23–32.

[9] 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.

[10] 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.

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

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

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

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

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

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

[34] J.-X. Pan and K.-T. Fang, "Maximum likelihood estimation," *Growth Curve Models and Statistical Diagnostics*, pp. 77–158, 2002.

[35] 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.

[36] 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.

[37] 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.

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

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.