Semi-supervised Autoencoder for Defect Prediction

James Hoang
School of Information System
vdthoang.2016@smu.edu.sg

ABSTRACT

Software defect prediction help developers to find bugs and prioritize their testing efforts. The traditional approaches focus on manually designing features encoding the characteristics of programs and employing different machine algorithms to construct defect prediction models. However, the manual defect features often fail to capture the semantic differences of programs, hence the prediction models may not be accurate. In this paper, we propose bridging the lexical gap between programs' semantics and defect prediction features by projecting natural language statements and programming element. Specifically, we introduce semi-supervised model inspired by autoencoder to learn semantic representation of source code as well as detect program's defect. Our framework is not only learn semantic features from token vectors extracted from programs' Abstract Syntax Trees (ASTs), but also construct classification model to detect program elements containing future bug. The experimental results show that our approach significantly outperforms the traditional approaches in defect prediction problem.

Keywords

Deep learning, defect prediction, bug localization, autoencoder, semi-supervised.

1. INTRODUCTION

Software defect prediction techniques [8, 11, 40] 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 use 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: first direction focuses on manually designing a set of features so that it can represent defects more effectively; the second direction focuses on building 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. [19] features focus on a complexity measure for the program elements, CK features [3] based on function and inheritance counts to understand the development of software projects, whereas MOOD features [7] tried to provide an overall assessment of a software system. The other features are constructed based on source code changes like, number of lines of code added, removed, etc. [11, 4]. On the other hand, many machine learning algorithm have been widely used for software defect prediction, including decision tree, logistic regression, Naive Bayes, etc [12]. However, the 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. [34] employed Deep Belief Network (DBN) [10] 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 [34] build semantic features and defect prediction model independently. Typically, the semantic features only learn from source files without considering the true label of this program element, hence the defect prediction model may not optimize. To tackle this problem, we propose a semi-supervised autoencoder allowing to extract semantic features and optimize the prediction model in one stage. Our proposed framework takes advantage of antoencoder [28] to construct semi-supervised learning model.

This paper makes the following contributions:

- We propose to leverage a powerful representation learning algorithm, namely deep learning, to learn semantic features from token vectors extracted from programs' ASTs automatically and use these features to optimize defect prediction models.
- Our evaluation results on 28 open source Java projects shows that our approach significantly improve the performance of defect prediction by 5.4% compared to traditional approaches.

The rest of this paper is summarized as follows. Section 2 briefly presents the defect prediction problem and our semi-supervised learning autoencoder. Section 3 shows the experimental results of our approaches. Section 4 presents threat to validity. Section 5 and Section 6 describe the related work and conclusion of our paper.

2. PROPOSED APPROACH

2.1 Defect Prediction

Figure 1 presents the overall framework of file-level defect prediction. Typically, the defect prediction problem is solved by following two specific steps. The first step is to label the program elements as bug or clean based on post-release defects for each file and then we extract the traditional features

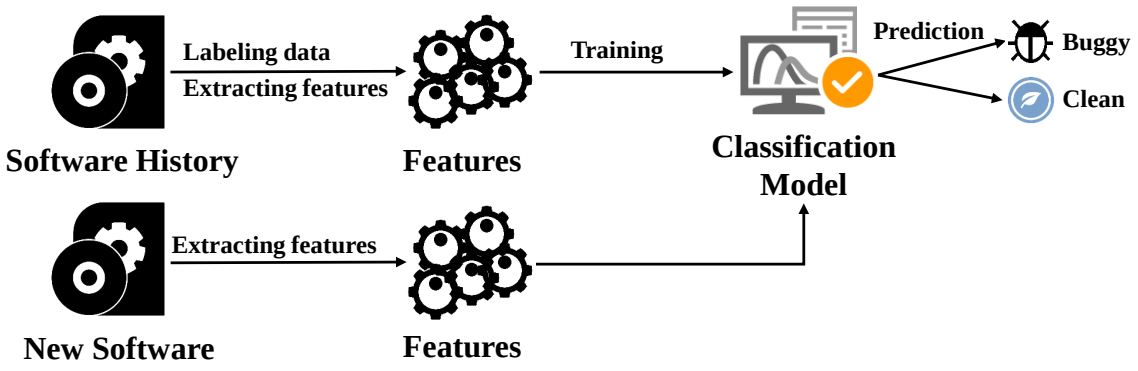

Figure 1: Defect Prediction Framework

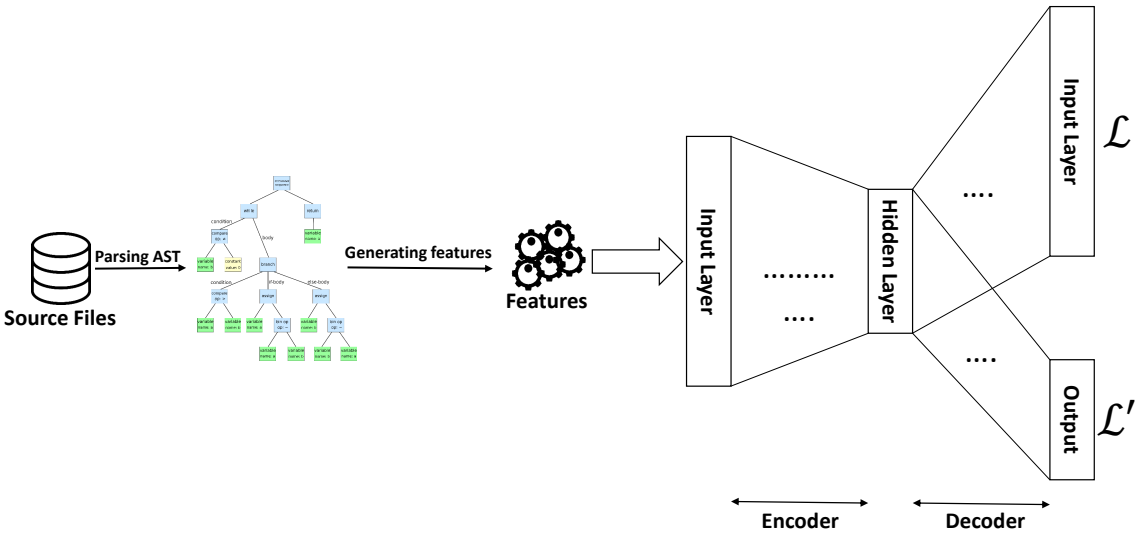

Figure 2: Semi-supervised AutoEncoder Framework

of these files. These features are briefly introduced in Section 5.1. The second step is to construct the classification model [2] used to predict whether a new program element contains bug or clean.

We refer to the software history used for building models as the training set, whereas the new software used to evaluate the trained models as the test set. The classification model are employed on test set to evaluate the performance of our defect prediction model.

2.2 Parsing Source Code and Generating Features

In our approach, we follow Wang et al. [34] to extract source code information to learn semantic features. Typically, the syntactic information from source code is collected based on Java Abstract Syntax Tree (AST) [27]. For each program element, we extract a vector of tokens of the three types of AST nodes: 1) nodes of method invocations and class instance creations, 2) declaration nodes, i.e., method declarations, type declarations, etc. and 3) control-flow nodes such as while statements, catch clauses, if statements, for statements, etc. Note that our semi-supervised learning only takes numerical vectors as inputs, and the lengths of the input vectors are the same. Thus, we apply Wang approaches [34] to map between integers and tokens, and encode token vectors to integer vector. Note that our integer vectors may have different lengths, we append 0 to the integer vectors to make all the lengths consistent and equal to the length of the longest vector. We also note that adding zeros does not affect the results, since it is simply representation transformation to make the vectors acceptable by neural network [34].

2.3 Semi-supervised Autoencoder

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

Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ denotes the set of program elements of a software project and $\mathcal{Y} = \{y_1, y_2, \ldots, y_n\}$ represents the label of each program element, where n is the number of source files in our collection data. Note that the program element is labeled as 1 if it contains bug, otherwise it will be labeled as 0 which means that it cleans. The source files can be collected from some popular software projects (e.g, ant, camel, lucene, etc.) [1]. Unlike the traditional approaches [36, 34] that independently learn semantic features and construct defect prediction model. Our semi-

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

supervised autoencoder (SSA) combines two different tasks to optimize the classification model for defect prediction problem. Typically, we attempt to learn semantic features function $f : \mathcal{X} \longmapsto \mathcal{X}$ and predict function $f' : \mathcal{X} \longmapsto \mathcal{Y}$, $y_i \in \mathcal{Y} = \{0, 1\}$ indicates whether a source file $x_i \in \mathcal{X}$ contains a bug which can be obtained by investigating software commit logs and bug report descriptions [6]. These two functions f and f' can be learned by minimizing the following objective function:

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

where $\mathcal{L}(\cdot,\cdot)$ and $\mathcal{L}'(\cdot,\cdot)$ are the empirical loss of semantic features and predict functions for the defect prediction problem, respectively. $\Omega(f, f')$ is the regularization terms imposing on the semantic and prediction functions. The trade-off between empirical loss and regularization terms are balanced by λ.

The overall framework of SSA are shown in Figure 2. The SSA model contains two four different parts: parsing abstract syntax tree, generating features, encoder, and decoder. The first two steps are briefly described in Section 2.2 to feed source files data to our deep neural network. Encoder and decoder are required to learn semantic features as well as defect prediction model. Note that our encoder and decoder steps are inspired by autoencoder [28] which is an unsupervised learning technique. However the original autoencoder only tries to learn the function $f : \mathcal{X} \longmapsto \mathcal{X}$ so that the output values $\hat{\mathcal{X}}$ are similar to input values $\mathcal{X}$. However, SSA attempts to learn semantic features and optimize defect prediction model, thus it takes into account of two functions, i.e., f and f' represent the semantic features and defect prediction respectively. According to Figure 2, our model tries to optimze two different loss functions, i.e., $\mathcal{L}$ and $\mathcal{L}'$ to optimize the defect prediction model. In encoder and decoder steps, we employ a fully connected neural network to fuse middle-level features extracted from source files to generate semantic features, where our network is learn to facilitate the determination on whether the given source code file is related to the given bug report based on the semantic features.

2.4 Imbalanced Problem in Defect Prediction

In most cases of defect prediction, only few source code files contain bug and a large number of source code files are *clean* [13], hence the imbalanced nature of this type of data increases the learning difficulty of this problem. For this reason, class imbalance learning specializes in tackling classification problems with imbablanced data is helpful for defect prediction problem [35].

To address this problem, we propose to learn the semantic features that may counteract the negative influence of the imbablanced data in the subsequent learning of defect prediction function. Inspired by [39], we introduce an unequal misclassification cost accroding to the imbalance ratio and train the fully connected network in a cost-sensitive manner.

Let r_n denote the ratio cost of incorrectly associating an *clean* source code file to a bug program element and r_p denote the cost of missing a buggy source code file in the training data. The weight of the semi-supervised autoencoder (SSA) networks $\mathcal{W}$ can be learned by minimizing the following objective function following Adam optimization [15].

$$\min_{\mathcal{W}} \sum_i \mathcal{L}(f(x_i), x_i) + r_n\mathcal{L}'(x_i, y_i; \mathcal{W})y_i + r_p\mathcal{L}'(x_i, y_i; \mathcal{W})(1 - y_i) + \lambda\|\mathcal{W}\|^2 \tag{2}$$

where $\mathcal{L}$ and $\mathcal{L}'$ are the loss function for semantic features and defect prediction model, respectively. λ is the trade-off parameter.

2.5 Setting for Training Semi-supervised Autoencoder

Some deep learning application [10, 28] reports an effective of deep learning models need well-tuned parameters, i.e., 1) the number of hidden layers, 2) the number of nodes in each hidden layer, and 3) the number of iterations. In [34], the authors pointed out that the deep learning model was optimized when they chose 10, 100, 200 as the number of hidden layers, number of nodes in each hidden layer, and number of iterations respectively. For the fair comparison with [34], we use these parameters to train our semi-supervised autoencoder model.

3. EXPERIMENTAL RESULTS

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

3.1 Results

This section presents our experimental results. We focus on the performance of our propose approaches, i.e., semi-supervised autoencoder (SSA), and answer the following research question: Does SSA outperform the two baselines: semantic features and traditional features?

To answer this question, we use two different features to build defect prediction models. We run the experiments on 28 sets of software project, each of which uses two versions of the same project (see Table **??**). 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 **??** shows the precision, recall and F1 of the defect prediction experiments. On average, code features achieve a F1 of 0.354, the semantic features constructed following [34] approaches achieve a F1 of 0.456, and our semi-supervised autoencoder (SSA) achieves a F1 of 0.510. The results demonstrate that we can improve the defect prediction F1 by 5.4% on average on 28 software projects.

We also use code features and semantic features separately to build defect prediction models by using two alternative classification algorithm, i.e., logistic regression and Naive Bayes. Table **??** shows the precision, recall and F1 scores on SSA vs. two different defect prediction models constructed by code features and semantic features using logistic regression. On average, code features and semantic features achieve F1 of 0.421 and 0.432 respectively. It shows that our SSA model improve the performance of F1 by 8.9% and 7.8% compared to two defect prediction models built using logistic regression algorithm. Table **??** presents the results of defect prediction models code features and semantic features using Naive Bayes algorithm, and SSA approaches. We see that our SSA outperforms these two baseline approaches. Table **??** shows the F1 results of our approach compared

Table 1: 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	DeepSemi	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
orientech	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
tananaev	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 2: 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
		DeepSemi	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
orientech	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
tananaev	nuvolabase	44.2 36.4 **40.0**	27.1 41.5 32.8	
nuvolabase	orientech	57.1 16.7 **25.8**	16.2 31.9 21.5	32.9 47.9 39.0
tananaev	orientech	25.4 43.1 **31.9**	18.5 45.1 26.3	
nuvolabase	tananaev	13.9 85.0 **23.9**	16.7 15.0 15.8	33.9 75.0 46.7
checkstyle	tananaev	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 3: Time cost used to construct deep semi-supervised learning model.

Project	Time (s)
checkstyle	10.2
nuvolabase	62.5
orientech	59.2
tananaev	5.67
Average	34.4

to the two code features and semantic features constructed be three different machine learning algorithms, i.e., decision tree, logistic regression, and Naive Bayes. It shows that SSA outperforms the other approaches on 28 software projects in average.

4. 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.).

5. RELATED WORK

5.1 Defect Prediction

The software defect prediction has been studied in the past decade [26, 21, 20, 40, 11, 25, 29, 33]. However, the traditional approaches in defect prediction often manually extract features from historical defect data to construct machine learning classification model [21]. McCabe et al. [19] introduced a graph-theoretic complexity measure for the control program elements which can be considered as a feature in defect prediction. CK features [3] focused on understanding of software development process, while MOOD features [7] 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. [23] 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. [25] 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. [17] 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 [11] 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. [5] estimated the capability of Support Vector Machine (SVM) [32] 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) [18] to predict the amount of residual faults of a software product. Khoshgoftaar et al. [14] showed that the Tree-based machine learning algorithms are efficiently in defect detection. Jing et al. [12] 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.

5.2 Deep Learning in Software Engineering

Recently, deep learning algorithms have been widely used to improve research tasks in software engineering. Lam et al. [16] combined deep neural network (DNN) [9] with rVSM [38], a revised vector space model, to improve the performance of bug localization. Raychev et al. [31] o reduced the problem of code completion to a natural-language processing problem of predicting probabilities of sentences and used recurrent neural network [22] to predict the probabilities of the next token. Mou et al. [24] 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. [30] employed recurrent neural network to build malware classification model in software system. Yuan et al. [37] adopted deep belief network (DBN) [10] 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. [36] 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. [34] also employed DBN to learn semantic features from source code. However, the existing features were extracted from abstract syntax tree since [33] claimed that Yang features [36] 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 semi-supervised learning autoencoder allowing to extract semantic features and construct classification model for defect prediction. We evaluate the effectiveness of our proposed approaches against Wang approaches [33] and the traditional machine learning algorithms (i.e., naive bayes, logistic regression, and random forest).

6. CONCLUSION

Our paper presents a semi-supervised autoencoder to learn semantic features as well as 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 28 software projects shows that our approaches could significantly improve the performance of defect prediction compared to two traditional approaches, i.e., code 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.

7. 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.

8. 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*, pages 215–226. IEEE, 2003.

[2] C. M. Bishop. Pattern recognition. *Machine Learning*, 128:1–58, 2006.

[3] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[4] F. B. e Abreu and R. Carapuça. Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Systems and Software*, 26(1):87–96, 1994.

[5] K. O. Elish and M. O. Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649–660, 2008.

[6] 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*, pages 23–32. IEEE, 2003.

[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*, 24(6):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*, pages 78–88. IEEE Computer Society, 2009.

[9] R. Hecht-Nielsen et al. Theory of the backpropagation neural network. *Neural Networks*, 1(Supplement-1):445–448, 1988.

[10] G. E. Hinton. Deep belief networks. *Scholarpedia*, 4(5):5947, 2009.

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

[12] 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*, pages 414–423. ACM, 2014.

[13] 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*, volume 1, pages 137–144. IEEE, 2010.

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

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

[16] 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*, pages 476–481. IEEE, 2015.

[17] 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*, pages 311–321. ACM, 2011.

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

[19] T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[20] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33(1), 2007.

[21] 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*, 17(4):375–407, 2010.

[22] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur. Recurrent neural network based language model. In *Interspeech*, volume 2, page 3, 2010.

[23] 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*, pages 181–190. ACM, 2008.

[24] 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.

[25] 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*, pages 364–373. IEEE, 2007.

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

[27] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[28] A. Ng. Sparse autoencoder. *CS294A Lecture notes*, 72(2011):1–19, 2011.

[29] 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*, pages 932–935. ACM, 2011.

[30] 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*, pages 1916–1920. IEEE, 2015.

[31] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *ACM SIGPLAN Notices*, volume 49, pages 419–428. ACM, 2014.

[32] J. A. Suykens and J. Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.

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

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

[35] S. Wang and X. Yao. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability*, 62(2):434–443, 2013.

[36] 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*, pages 17–26. IEEE, 2015.

[37] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue. Droid-sec: deep learning in android malware detection. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 371–372. ACM, 2014.

[38] 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*, pages 14–24. IEEE Press, 2012.

[39] Z.-H. Zhou and X.-Y. Liu. Training cost-sensitive neural networks with methods addressing the class imbalance problem. *IEEE Transactions on Knowledge and Data Engineering*, 18(1):63–77, 2006.

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