Deep Learning for Software Defect Prediction: A Survey

Safa Omri
Karlsruhe Institute of Technology
Karlsruhe, Germany
safa.omri@kit.edu

Carsten Sinz
Karlsruhe Institute of Technology
Karlsruhe, Germany
carsten.sinz@kit.edu

ABSTRACT

Software fault prediction is an important and beneficial practice for improving software quality and reliability. The ability to predict which components in a large software system are most likely to contain the largest numbers of faults in the next release helps to better manage projects, including early estimation of possible release delays, and affordably guide corrective actions to improve the quality of the software. However, developing robust fault prediction models is a challenging task and many techniques have been proposed in the literature. Traditional software fault prediction studies mainly focus on manually designing features (e.g. complexity metrics), which are input into machine learning classifiers to identify defective code. However, these features often fail to capture the semantic and structural information of programs. Such information is needed for building accurate fault prediction models. In this survey, we discuss various approaches in fault prediction, also explaining how in recent studies deep learning algorithms for fault prediction help to bridge the gap between programs' semantics and fault prediction features and make accurate predictions.

KEYWORDS

deep learning, software testing, software defect prediction, machine learning, software quality assurance

ACM Reference Format:
Safa Omri and Carsten Sinz. 2020. Deep Learning for Software Defect Prediction: A Survey. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), October 5–11, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3387940.3391463

1 INTRODUCTION

Nowadays, software quality assurance is overall the most expensive activity for nearly all software developing companies [43], since team members need to spend a significant amount of their time inspecting the entire software in detail rather than, for example, implementing new features. Software quality assurance activities, such as source code inspection, assist developers infi nding potential bugs and allocating their testing efforts. They have a great influence on producing high quality reliable software. Numerous research studies have analyzed software fault prediction techniques to help prioritize software testing and debugging. Software fault prediction is a process of building classifiers to anticipate which software modules or code areas are most likely to fail. Most of these techniques focus on designing features (e.g. complexity metrics) that correlate with potentially defective code. Object-oriented metrics were initially suggested by Chidamber and Kemerer [7]. Basili et al. [3] and Briand et al. [5] were among thefi rst to use such metrics to validate and evaluate fault-proneness. Subramanyam and Krishnan [44] and Tang et al. [46] showed that these metrics can be used as early indicators of externally visible software quality. D'Ambros et al. have compared popular fault prediction approaches for software systems [10], namely, process metrics [31], previous faults [24] and source code metrics [3]. Nagappan et al. [34] presented empirical evidence that code complexity metrics can predict post-release faults. Our previous work [37] takes into consideration not only code complexity metrics but also the faults detected by static analysis tools to build accurate pre-release fault predictors. Numerous research studies have analyzed code churn (number of lines of code added, removed, etc.) as a variable for predicting faults in large software systems [21, 33, 38]. All these research studies have gone into carefully designing features which are able to discriminate defective code from non-defective code such as code size, code complexity (e.g. Halstead, McCabe, CK features), code churn metrics (e.g. the number of code lines changed), or process metrics. Most defect prediction approaches consider defect prediction as a binary classification problem that can be solved by classification algorithms, e.g., Support Vector Machines (SVM), Naive Bayes (NB), Decision Trees (DT), or Neural Networks (NN). Such approaches simply classify source code changes into two categories: fault-prone or not fault-prone.

Those approaches, however, do not sufficiently capture the syntax and different levels of semantics of source code, which is an important capability for building accurate prediction models. Specifically, in order to make accurate predictions, features need to be discriminative: capable of distinguishing one instance of code region from another. The existing traditional features cannot distinguish code regions with different semantics but similar code structure. For example, in Figure 1, there are two Javafi les, both of which contain a `for` statement, a `remove` function and an `add` function. The only difference between the twofi les is the order of the `remove` and `add` function. *File2.java* will produce a `NoSuchElementException` when the function is called with an empty queue. Using traditional features to represent these twofi les, their feature vectors are identical, because these twofi les have the same source code characteristics in terms of lines of code, function calls, raw programming tokens, etc. However, the semantic content is different. Features that can distinguish such semantic differences should enable the building of more accurate prediction models. To bridge the gap between programs' semantic information and features used for defect prediction, some approaches propose to leverage a powerful representation-learning

ICSEW'20, October 5–11, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3391463

algorithm, namely deep learning, to capture the semantic representation of programs automatically and use this representation to improve defect prediction.

```
static void function (Queue myQueue) {
    int i;
    for (i = 0; i < 10; i++) {
        // insert i to the tail of the queue
        myQueue.add(i);
        // remove the head of the queue
        myQueue.remove();
    }
}
```

File1.java

```
static void function (Queue myQueue) {
    int i;
    for (i = 0; i < 10; i++) {
        // remove the head of the queue
        myQueue.remove();
        // insert i to the tail of the queue
        myQueue.add(i);
    }
}
```

File2.java

Figure 1: A motivating example: `File2.java` will exhibit an exception when the function is called with an empty queue.

In this survey, we review the different deep learning technologies used in software quality assurance to predict faults, and provide a survey on the state-of-the-art in deep learning methods applied to software defect prediction.

2 SOFTWARE DEFECT PREDICTION PROCESS

Fault prediction is an active research area in thefi eld of software engineering. Many techniques and metrics have been developed to improve fault prediction performance. In recent decades, numerous studies have examined the realm of software fault prediction. Figure 2 briefly shows the history of software fault prediction studies in about the last 20 years.

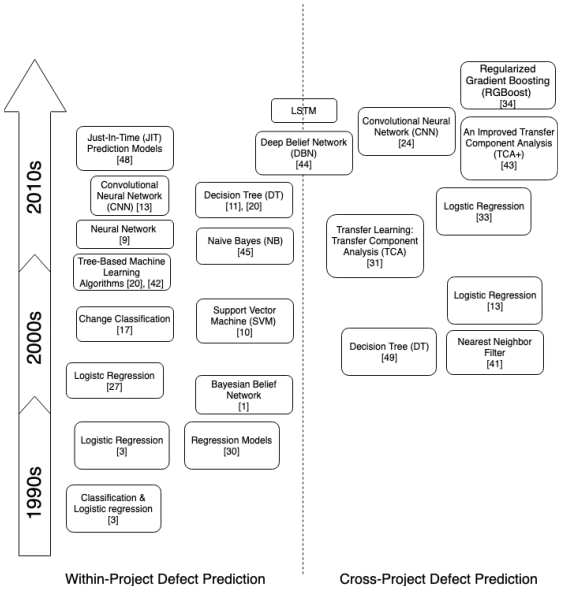

Figure 2: History of Software Defect Prediction

As the process shows in Figure 3, thefi rst step is to collect source code repositories from software archives. The second step is to extract features from the source code repositories and the commits contained therein. There are many traditional features defined in past studies, which can be categorized into two kinds:

Metric	Formula	Interpretation
Accuracy	$\frac{TP+TN}{TP+TN+FP+FN}$	Overall performance of model
Precision	$\frac{TP}{TP+FP}$	How accurate the positive predictions are
Recall	$\frac{TP}{TP+FN}$	Coverage of actual positive sample
F1 score	$\frac{2TP}{2TP+FP+FN}$	Hybrid metric useful for unbalanced classes

Table 1: Common metrics used to assess the performance of classification models

code metrics (e.g., McCabe features and CK features) and process metrics (e.g., change histories). The extracted features represent the train and test dataset. To select the best-fit defect prediction model, the most commonly used method is called k-fold cross-validation that splits the training data into k groups to validate the model on one group while training the model on the $k-1$ other groups, all of this k times. The error is then averaged over the k runs and is named cross-validation error. The diagnostics of the model is based on these features: *(1) Bias*: the bias of a model is the difference between the expected prediction and the correct model that we try to predict for given data points. *(2) Variance*: the variance of a model is the variability of the model prediction for given data points. *(3) Bias/variance tradeoff*: the simpler the model, the higher the bias, and the more complex the model, the higher the variance. Figure 4 shows a brief summary of how underfitting, overfitting and a suitablefi t looks like for the three commonly used techniques regression, classification and deep learning. Once the model has been chosen, it is trained on the entire dataset and tested on the test dataset. Most defect prediction approaches take defect prediction as a binary classification problem. Afterfi tting the models, the test data is fed into the trained classifier (the best-fit prediction model), which can predict whether thefi les are buggy or clean. Afterwards, in order to assess the performance of the selected model, quality metrics are computed. To have a more complete picture when assessing the performance of a model, a confusion matrix is used. It is defined as shown in Figure 5. We summarize the metrics for the performance of classification models in Table 1.

2.1 Within-Project Defect Prediction

Within-project defect prediction uses training data and test data that are from the same project. Many machine learning algorithms have been adopted for within-project defect prediction, including Support Vector Machines (SVM) [12], Bayesian Belief Networks [1], Naive Bayes (NB) [53], Decision Trees (DT) [13], [22], [49], Neural Networks (NN) [11], or Dictionary Learning [17]. Elish et al. [12] evaluated the feasibility of SVM in predicting defect-prone software modules, and they compared SVM against eight statistical and machine learning models on four NASA datasets. Their results showed that SVM is generally better than, or at least competitive with other models, e.g., Logistic Regression, Bayesian techniques, etc. Amasaki et al. [1] used a Bayesian Belief Network to predict thefi nal quality of a software product. They evaluated their approach on a closed project, and the results showed that their proposed method can

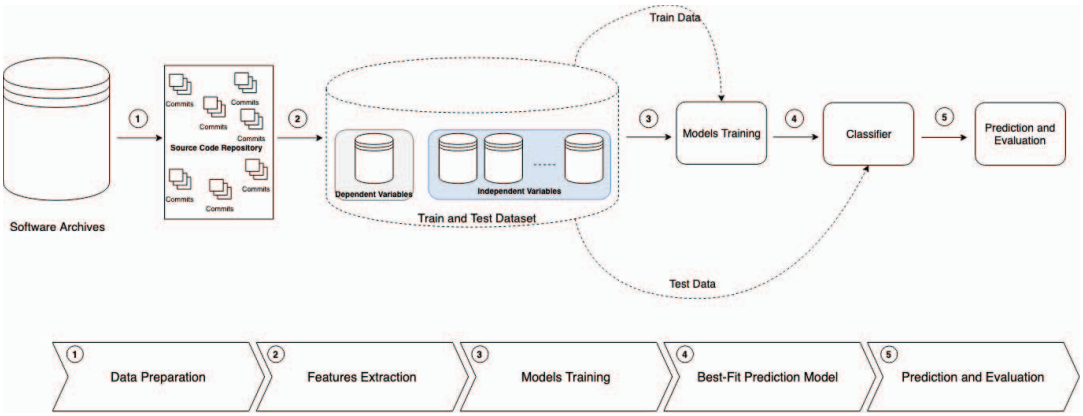

Figure 3: Software Defect Prediction Process

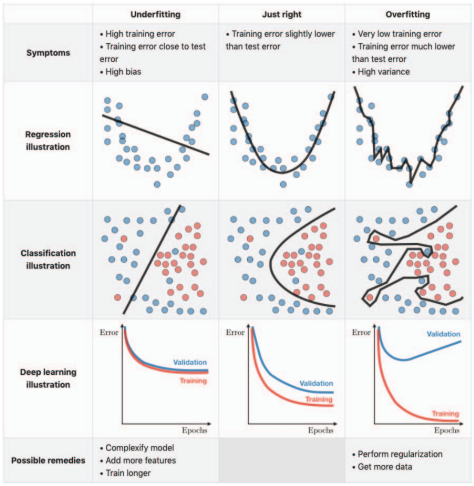

Figure 4: Fitting Model Diagnostics [2]

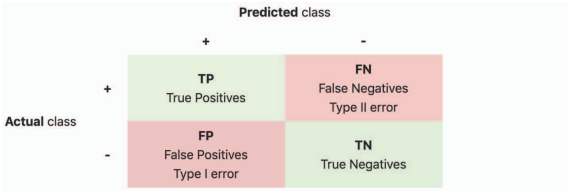

Figure 5: Confusion Matrix

predict bugs that the Software Reliability Growth Model (SRGM) cannot handle. Wang et al. [49] and Khoshgoftaar et al. [22] examined the performance of tree-based machine learning algorithms on defect prediction. Their results indicate that tree-based algorithms can generate good predictions. Tao et al. [53] proposed a Naive Bayes based defect prediction model, and they evaluated the proposed approach on 11 datasets from the PROMISE defect data repository. Their experimental results showed that the Naive Bayes based defect prediction models could achieve better performance than *J*48 (decision tree) based prediction models. Jing et al. [17] introduced the dictionary learning technique to defect prediction. Their cost-sensitive dictionary learning based approach could significantly improve defect prediction in their experiments. Wang et al. [52] used a Deep Belief Network (DBN) to generate semantic features forfi le-level defect prediction tasks. In Wang et al.'s work [52], to evaluate the performance of DBN-based semantic features as well as traditional features, they built prediction models by using three typical machine learning algorithms, i.e., ADTree, Naive Bayes, and Logistic Regression. Their experimental results show that the learned DBN-based semantic features consistently outperform the traditional defect prediction features on these machine learning classifiers. Most of the above approaches are designed forfi le-level defect prediction. For change-level defect prediction, Mockus and Weiss [30] and Kamei et al. [19] predicted the risk of a software change by using change measures, e.g., the number of subsystems touched, the number offi les modified, the number of added lines, and the number of modification requests. Kim et al. [23] used the identifiers in added and deleted source code and the words in change logs to classify changes as being fault-prone or not fault-prone. Jiang et al. [16] and Xia et al. [54] built separate prediction models with characteristic features and meta features for each developer to predict software defects in changes. Tan et al. [45] improved change classification techniques and proposed online defect prediction models for imbalanced data. Their approach uses time sensitive change classification to address the incorrect evaluation introduced by cross-validation. McIntosh et al. [28] studied the performance of change-level defect prediction as software systems evolve. Change classification can also predict whether a commit is buggy or not [39], [41], [14]. In Wang et al.'s work [52], they also compare the DBN-based semantic features with the widely used change-level defect prediction features, and ther results suggest that the DBN-based semantic features can also outperform change-level features.

However, sufficient defect data is often unavailable for many projects and companies. This raises the need for cross-project bug localization, i.e., the use of data from one project to help locate bugs in another project.

2.2 Cross-Project Defect Prediction

Due to the lack of data, it is often difficult to build accurate models for new projects. Recently, more and more papers studied the *cross-project defect prediction* problem, where the training data and test data come from different projects. Some studies ([25], [29], [57]) have been done on evaluating cross-project defect prediction against within-project defect prediction and show that cross-project defect prediction is still a challenging problem. He et al. [15] showed the feasibility tofi nd the best cross-project models among all available models to predict defects on specific projects. Turhan et al. [48] proposed a nearest-neighborfi lter to improve cross-project defect prediction. Zimmermann et al. [57] evaluated the performance of cross-project defect prediction on 12 projects and their 622 combinations. They found that the defect prediction models at that time could not adapt well to cross-project defect prediction. Li et al. [26] proposed defect prediction via convolutional neural networks (DP-CNN). Their work differs from the above-mentioned approaches in that they utilize deep learning technique (i.e., CNN) to automatically generate discriminative features from source code, rather than manually designing features which can capture semantic and structural information of programs. Their features lead to more accurate predictions. The state-of-the-art cross-project defect prediction is proposed by Nam et al. [35], who adopted a state-of-the-art transfer learning technique called Transfer Component Analysis (TCA). They further improved TCA as TCA+ by optimizing TCA's normalization process. They evaluated TCA+ on eight open-source projects, and the results show that their approach significantly improves cross-project defect prediction. Xia et al. [54] proposed HYDRA, which leverages a genetic algorithm and ensemble learning (EL) to improve cross-project defect prediction. HYDRA requires massive training data and a portion (5%) of labeled data from test data to build and train the prediction models. TCA+ [35] and HYDRA [54] are the two state-of-the-art techniques for cross-project defect prediction. However, in Wang et al.'s work [51], they only use TCA+ as baseline for cross-project defect prediction. This is because HYDRA requires that the developers manually inspect and label 5% of the test data, while in real-world practice, it is very expensive to obtain labeled data from software projects, which requires the developers' manually inspection, and the ground truth might not be guaranteed. Most of the above existing cross-project approaches are examined forfi le-level defect prediction only. Recently, Kamei et al. [18] empirically studied the feasibility of change level defect prediction in a cross-project context. Wang et al. [51] examines the performance of Deep Belief Network (DBN)-based semantic features on change-level cross-project defect prediction tasks. The main differences between this and existing approaches for within-project defect prediction and cross-project defect prediction are as follows. First, existing approaches to defect prediction are based on manually encoded traditional features which are not sensitive to the programs' semantic information, while Wang et al.'s approach automatically learns the semantic features using a DBN and uses these features to perform defect prediction tasks. Second, since Wang et al.'s method requires only the source code of the training and test projects, it is suitable for both within-project and cross-project defect prediction. The semantic features can capture the common characteristics of defects, which implies that the semantic features trained from one project can be used to predict a different project, and thus is applicable in cross-project defect prediction.

Deep learning-based approaches require only the source code of the training and test projects, and are therefore suitable for both within-project and cross-project defect prediction. In the next session, we explain, based on recent research, how effective and accurate fault-prediction models developed using deep learning techniques are.

3 DEEP LEARNING IN SOFTWARE DEFECT PREDICTION

Recently, deep learning algorithms have been adopted to improve research tasks in software engineering. The most popular deep learning techniques are: Deep Belief Networks (DBN), Recurrent Neural Networks, Convolutional Neural Networks and Long Short Term Memory (LSTM), see Table 2. Yang et al. [56] propose an approach that leverages deep learning to generate new features from existing ones and then use these new features to build defect prediction models. Their work was motivated by the weaknesses of logistic regression (LR), which is that LR cannot combine features to generate new features. They used a Deep Belief Network (DBN) to generate features from 14 traditional change level features, including the following: number of modified subsystems, modified directories, modifiedfi les, code added, code deleted, lines of code before/after the change,fi les before and after the change, and several features related to developers' experience [56]. The work of Wang et al. [51] differs from the above study mainly in three aspects. First, they use a DBN to learn semantic features directly from source code, while Yang et al. use relations among existing features. Since the existing features cannot distinguish between many semantic code differences, the combination of these features would still fail to capture semantic code differences. For example, if two changes add the same line at different locations in the samefi le, the traditional features cannot distinguish between the two changes. Thus, the generated new features, which are combinations of the traditional features, would also fail to distinguish between the two changes.

How to explain deep learning results is still a challenging question in the AI community. To interpret deep learning models, Andrej et al. [20] used character level language models as an interpretable testbed to explain the representations and predictions of a Recurrent Neural Network (RNN). Their qualitative visualization experiments demonstrate that RNN models could learn powerful and often interpretable long-range interactions from real-world data. Radford et al. [42] focus on understanding the properties of representations learned by byte-level recurrent language models for sentiment analysis. Their work reveals that there exists a sentiment unit in the well-trained RNNs (for sentiment analysis) that has a direct influence on the generative process of the model. Specifically, simply fixing its value to be positive or negative can generate samples with the corresponding positive or negative sentiment. The above studies show that to some extent deep learning models are interpretable. However, these two studies focused on interpreting RNNs on text analysis. Wang et al. [51] leverages a different deep learning model, Deep Belief Networks (DBN), to analyze the ASTs of source code. The DBN adopts different architectures and learning processes from

Techniques	Definition	Advantages	Drawbacks	Ref.
RNN	RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being depended on the previous computations.	- Possibility of processing input of any length - Model size not increasing with size of the input - Computation takes into account historical information	- Slow computation - Difficulty of accessing information from a long time ago - Cannot consider any future input for the current state	[50]
LSTM	A long short-term memory (LSTM) network is a type of RNN model that avoids the vanishing gradient problem by adding 'forget' gates.	- Remembering information for a long periods of time	- It takes longer to train - It requires more memory to train	[8], [9]
CNN	CNN is a class of deep neural network, it uses convolution in place of general matrix multiplication in at least one of their layers.	- It automatically detects the important features without any human supervision.	- need a lot of training data. - High computational cost.	[26], [32], [40]
Stacked Auto-Encoder	A stacked autoencoder is a neural network consist several layers of sparse autoencoders where output of each hidden layer is connected to the input of the successive hidden layer.	- Possible use of pre-trained layers from another model, to apply transfer learning - It does not require labeled inputs to enable learning	- Computationally expensive to train - Extremely uninterpretable - The underlying math is more complicated - Prone to overfitting, though this can be mitigated via regularization	[27], [47]
DBN	DBN is an unsupervised probabilistic deep learning algorithm.	- Only needs a small labeled dataset - It is a solution to the vanishing gradient problem	- It overlooks the structural information of programs	[52]
Logistic Regression	LR is used to describe data and to explain the relationship between one dependent binary variable and independent variables.	- Easy to implement - Very efficient to train	- It cannot combine different features to generate new features. - It performs well only when input features and output labels are in linear relation	[19]
SVM	SVM is a supervised learning model. It can be used for both regression and classification tasks.	- Using different kernel function it gives better prediction result - Less computation power	- Not suitable for large number of software metrics	[12]
Decision Tree	DT is a decision support tool that uses a tree-like graph or model of decisions and their possible consequences.	Tree based methods empower predictive models with high accuracy, stability and ease of interpretation.	- Construction of decision tree is complex	[13], [22], [49]

Table 2: Common machine learning and deep learning techniques used in software defect prediction

RNNs. For example, an RNN (e.g., LSTM) can, in principle, use its memory cells to remember long-range information that can be used to interpret data it is currently processing, while a DBN does not have such memory cells. Thus, it is unknown whether DBN models share the same properties (w.r.t interpretability) as RNNs. Many studies used a topic model [4] to extract semantic features for different tasks in software engineering ([6], [36], [55]). Nguyen et al. [36] leveraged a topic model to generate features from source code for within-project defect prediction. However, their topic model handles each sourcefi le as an unordered token sequence. Thus, the generated features cannot capture structural information in a sourcefi le. A just-in-time defect prediction technique was proposed by Kamei et al. which leverages the advantages of Logistic Regression (LR) [19]. However, logistic regression has two weaknesses. First, in logistic regression, the contribution of each feature is calculated independently, which means that LR cannot combine different features to generate new ones. For example, given two features x and y, if $x \times y$ is a highly relevant feature, it is not enough to input only x and y because logistic regression cannot generate the new feature $x \times y$. Second, logistic regression performs well only when input features and output labels are in linear relation. Due to these two weaknesses, the selection of input features becomes crucial when using logistic regression. The bad selection of features may result in a non-linear relation for output labels, leading to bad training performance or even training failure. This severe problem leads some studies to adopt Deep Belief Network (DBN), which is one of the state-of-the-art deep learning approaches. The biggest advantage of DBN, as shown in Table 2, over logistic regression is that DBNs can generate a more expressive feature set from the initial feature set. We summarizes in Table 2 the most commonly used machine learning and deep learning techniques in software defect prediction.

4 CONCLUSION

With the ever-increasing scale and complexity of modern software, software reliability assurance has become a significant challenge. To enhance the reliability of software, we consider predicting potential code defects in software implementations a beneficial direction, which has the potential to dramatically reduce the workload of software maintenance. Specifically, we see the highest potential in a defect prediction framework which utilizes deep learning algorithms for automated feature generation from source code with the semantic and structural information preserved. Moreover, our survey corroborates the feasibility of deep learning techniques in thefi led of program analysis.

REFERENCES

[1] Sousuke Amasaki, Yasunari Takagi, Osamu Mizuno, and Tohru Kikuno. 2003. A Bayesian Belief Network for Assessing the Likelihood of Fault Content. In *Proceedings of the 14th International Symposium on Software Reliability Engineering.*

[2] Afshine Amidi. 2018. *cheatsheet-machine-learning-tips-and-tricks.* https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-machine-learning-tips-and-tricks

[3] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. 1996. A Validation of Object-Oriented Design Metrics As Quality Indicators. *IEEE Trans. Softw. Eng.* (1996).

[4] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. *J. Mach. Learn. Res.* (2003).

[5] Lionel C. Briand, Jürgen Wüst, Stefan V. Ikonomovski, and Hakim Lounis. 1999. Investigating Quality Factors in Object-oriented Designs: An Industrial Case Study. In *Proceedings of the 21st International Conference on Software Engineering*.

[6] Tse-Hsun Chen, Stephen W. Thomas, Meiyappan Nagappan, and Ahmed E. Hassan. 2012. Explaining Software Defects Using Topic Models. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*.

[7] S. R. Chidamber and C. F. Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng.* (1994).

[8] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. 2019. Lessons Learned from Using a Deep Tree-Based Model for Software Defect Prediction in Practice. In *Proceedings of the 16th International Conference on Mining Software Repositories*.

[9] Khanh Hoa Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya K. Ghose, Taeksu Kim, and Chul-Joo Kim. 2018. A deep tree-based model for software defect prediction. *ArXiv* (2018).

[10] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2012. Evaluating Defect Prediction Approaches: A Benchmark and an Extensive Comparison. *Empirical Softw. Engg.* (2012).

[11] Elhampaikari, Michael M.richter, and Guentherruhe. 2012. Defect prediction using case-based reasoning: an attribute weighting technique based upon sensitivity analysis in neural network. *International Journal of Software Engineering and Knowledge Engineering* (2012).

[12] Karim O. Elish and Mahmoud O. Elish. 2008. Predicting Defect-Prone Software Modules Using Support Vector Machines. *J. Syst. Softw.* (2008).

[13] N. Gayatri, Nickolas Savarimuthu, and A. Reddy. 2010. Feature Selection Using Decision Tree Induction in Class level Metrics Dataset for Software Defect Predictions. *Lecture Notes in Engineering and Computer Science* (2010).

[14] Andrew Habib and Michael Pradel. 2019. Neural Bug Finding: A Study of Opportunities and Challenges. *CoRR* (2019).

[15] Z. He, F. Peters, T. Menzies, and Y. Yang. 2013. Learning from Open-Source Projects: An Empirical Study on Defect Prediction. In *ACM IEEE International Symposium on Empirical Software Engineering and Measurement*.

[16] Tian Jiang, Lin Tan, and Sunghun Kim. 2013. Personalized Defect Prediction. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*.

[17] Xiao-Yuan Jing, Shi Ying, Zhi-Wu Zhang, Shan-Shan Wu, and Jin Liu. 2014. Dictionary Learning Based Software Defect Prediction. In *Proceedings of the 36th International Conference on Software Engineering*.

[18] Yasutaka Kamei, Takafumi Fukushima, Shane Mcintosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E. Hassan. 2016. Studying Just-in-Time Defect Prediction Using Cross-Project Models. *Empirical Softw. Engg.* (2016).

[19] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A Large-Scale Empirical Study of Just-in-Time Quality Assurance. *IEEE Trans. Softw. Eng.* (2013).

[20] Andrej Karpathy, Justin Johnson, and Fei Fei Li. 2015. Visualizing and Understanding Recurrent Networks. *Cornell Univ. Lab.* (2015).

[21] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan. 1996. Detection of Software Modules with High Debug Code Churn in a Very Large Legacy System. In *Proceedings of the The Seventh International Symposium on Software Reliability Engineering*.

[22] Taghi M. Khoshgoftaar and Naeem Seliya. 2002. Tree-Based Software Quality Estimation Models For Fault Prediction. In *Proceedings of the 8th International Symposium on Software Metrics*.

[23] Sunghun Kim, E. James Whitehead, and Yi Zhang. 2008. Classifying Software Changes: Clean or Buggy? *IEEE Trans. Softw. Eng.* (2008).

[24] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. 2007. Predicting Faults from Cached History. In *Proceedings of the 29th International Conference on Software Engineering*.

[25] Barbara A. Kitchenham, Emilia Mendes, and Guilherme H. Travassos. 2007. Cross versus Within-Company Cost Estimation Studies: A Systematic Review. *IEEE Trans. Softw. Eng.* (2007).

[26] J. Li, P. He, J. Zhu, and M. R. Lyu. 2017. Software Defect Prediction via Convolutional Neural Network. In *IEEE International Conference on Software Quality, Reliability and Security (QRS)*.

[27] C. Manjula and Lilly Florence. 2019. Deep neural network based hybrid approach for software defect prediction using software metrics. *Cluster Computing* (2019).

[28] Shane McIntosh and Yasutaka Kamei. 2018. Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-in-Time Defect Prediction. In *Proceedings of the 40th International Conference on Software Engineering*.

[29] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayse Bener. 2010. Defect prediction from static code features: Current results, limitations, new approaches. *Autom. Softw. Eng.* (2010).

[30] A. Mockus and D. M. Weiss. 2000. Predicting risk of software changes. *Bell Labs Technical Journal* (2000).

[31] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. 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*.

[32] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*.

[33] Nachiappan Nagappan and Thomas Ball. 2005. Use of Relative Code Churn Measures to Predict System Defect Density. In *Proceedings of the 27th International Conference on Software Engineering*.

[34] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining Metrics to Predict Component Failures. In *Proceedings of the 28th International Conference on Software Engineering*.

[35] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. 2013. Transfer Defect Learning. In *Proceedings of the International Conference on Software Engineering*.

[36] Tung Thanh Nguyen, Tien N. Nguyen, and Tu Minh Phuong. 2011. Topic-Based Defect Prediction (NIER Track). In *Proceedings of the 33rd International Conference on Software Engineering*.

[37] S. Omri, P. Montag, and C. Sinz. 2018. Static Analysis and Code Complexity Metrics as Early Indicators of Software Defects. *Journal of Software Engineering and Applications* (2018).

[38] S. Omri, C. Sinz, and P. Montag. [n.d.]. An Enhanced Fault Prediction Model for Embedded Software based on Code Churn, Complexity Metrics, and Static Analysis Results. ICSEA 2019 : The Fourteenth International Conference on Software Engineering Advances.

[39] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*.

[40] Anh Phan, Le Nguyen, and Lam Bui. 2018. Convolutional Neural Networks over Control Flow Graphs for Software Defect Prediction. (2018).

[41] Lutz Prechelt and Alexander Pepper. 2014. Why Software Repositories Are Not Used for Defect-Insertion Circumstance Analysis More Often: A Case Study. *Inf. Softw. Technol.* (2014).

[42] Alec Radford, Rafal Jozefowicz, and Ilya Sutskever. 2017. Learning to Generate Reviews and Discovering Sentiment. (2017).

[43] R. Rana, M. Staron, J. Hansson, and M. Nilsson. 2014. Defect prediction over software life cycle in automotive domain state of the art and road map for future. In *9th International Conference on Software Engineering and Applications (ICSOFT-EA)*.

[44] Ramanath Subramanyam and M. S. Krishnan. 2003. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. *IEEE Trans. Softw. Eng.* (2003).

[45] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. Online Defect Prediction for Imbalanced Data. In *Proceedings of the 37th International Conference on Software Engineering*.

[46] Mei-Huei Tang, Ming-Hung Kao, and Mei-Hwa Chen. 1999. An Empirical Study on Object-Oriented Metrics. In *Proceedings of the 6th International Symposium on Software Metrics*.

[47] Haonan Tong, Bin Liu, and Shihai Wang. 2017. Software Defect Prediction Using Stacked Denoising Autoencoders and Two-stage Ensemble Learning. *Information and Software Technology* (2017).

[48] Burak Turhan, Tim Menzies, Ayundefinede B. Bener, and Justin Di Stefano. 2009. On the Relative Value of Cross-Company and within-Company Data for Defect Prediction. *Empirical Softw. Engg.* (2009).

[49] Jun Wang, Beijun Shen, and Yuting Chen. [n.d.]. Compressed C4.5 Models for Software Defect Prediction. In *Proceedings of the 2012, 12th International Conference on Quality Software*.

[50] Jinyong Wang and Ce Zhang. 2018. Software reliability prediction using a deep learning model based on the RNN encoder-decoder. *Reliab. Eng. Syst. Saf.* (2018).

[51] S. Wang, T. Liu, J. Nam, and L. Tan. 2018. Deep Semantic Feature Learning for Software Defect Prediction. *IEEE Transactions on Software Engineering* (2018).

[52] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically Learning Semantic Features for Defect Prediction. In *Proceedings of the 38th International Conference on Software Engineering*.

[53] T. Wang and W. Li. [n.d.]. Naive Bayes Software Defect Prediction Model. In *2010 International Conference on Computational Intelligence and Software Engineering*.

[54] X. Xia, D. Lo, X. Wang, and X. Yang. 2016. Collective Personalized Change Classification With Multiobjective Search. *IEEE Transactions on Reliability* (2016).

[55] Xihao Xie, Wen Zhang, Ye Yang, and Qing Wang. 2012. DRETOM: Developer Recommendation Based on Topic Models for Bug Resolution. In *Proceedings of the 8th International Conference on Predictive Models in Software Engineering*.

[56] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep Learning for Just-in-Time Defect Prediction. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security*.

[57] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. Cross-Project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT*.