Transfer Learning for Predicting Software Faults

Anh Phan Viet
Le Quy Don Technical University
Hanoi, Vietnam
anhpv@mta.edu.vn

Khanh Duy Tung Nguyen
Le Quy Don Technical University
Hanoi, Vietnam
tungkhanhmta@gmail.com

Lai V Pham
Military Science and Technology Institute
Hanoi, Vietnam
garry@cinnamon.is

***Abstract*—This paper investigates a transfer learning application for predicting software faults. Detecting faulty modules in software projects is challenging due to two main issues 1) the low quality of existing handcrafted features leads to the bad performance of traditional learning models and 2) the shortage of annotated data hinders applying deep neural networks. Recently, transfer learning is a good solution to train deep neural networks with insufficient data. Our experiments for tasks of within-project and cross-project software fault prediction have shown the transferable possibility among project data. As a result, the performance of the base model is significantly improved and achieves competitive results with the state of the art method.**

***Index Terms*—Within-project Prediction, Cross-project Prediction, Convolutional Neural Networks, Transfer Learning.**

I. Introduction

Timely predicting faulty modules in projects by analyzing source code brings many benefits to the software industry. This saves the effort of fixing bugs and therefore reduces the development time and software cost. A common approach for software fault prediction (SFP) is applying machine learning on handcrafted features called software metrics, [5]. A software metric is to evaluate some properties of code snippets or its specifications, e.g. the number of operators and operands. The biggest challenge of software metrics-based approaches is designing a set of good features with high relevance to the causes of software faults. Recently, deep learning has been investigated for source code analysis problems due to the ability to automatically learn programs' features [12]. However, deep learning requires a huge amount of data for training the predictive models. This requirement is an obstacle when adapting deep learning to software fault prediction because of limited annotated data. To overcome the obstacle, this paper utilizes transfer learning techniques to leverage the data among different software projects.

Nowadays, deep neural networks with transfer learning have made many breakthroughs in many domains including computer vision and natural language processing (NLP) [4], [14]. Automatically learning high-level features is a highlight of deep neural networks, but the training process needs a large amount of data to be converged. Transfer learning leverages the data from other sources to pretrain the model, and then fine-tune on the current problem with limited data. The efficiency of transfer learning can be illustrated via BERT [4] and XLNet [14]. Due to the integration of transformers into pretraining, the two models reached state-of-the-art performance on a wide range of NLP problems including sentiment analysis, document ranking, question answering, and natural language inference.

The motivation for utilizing transfer learning for software fault prediction problems comes from two main reasons 1) limited data and 2) the transferable knowledge among projects. Firstly, collecting a large number of source files with corresponding faults is difficult in practice. For this reason, most of the datasets for SFP problems are limited and imbalanced [15]. Secondly, different projects can have common mistakes in programming languages such as null pointer dereference, buffer overflow, and access violations. Therefore, transfer learning can be a good solution to overcome the obstacles of SFP problems.

In summary, the main contributions of the paper can be presented as follows:

- Investigating the transfer learning to deal with data shortage in software-fault prediction.
- Proposing two scenarios to leverage software data among projects, and experimentally proving the efficiency on two main tasks being predicting faults in a project and among projects.
- Applying an unsupervised learning algorithm to train vector representations for assembly instructions.

The rest of the paper is structured as follows: Section II surveys studies related to software fault prediction and transfer learning. The based model architecture and scenarios in adapting transfer learning are specified in Section III. Section IV presents the baselines and hyperparameter settings. Section V discusses the experimental results and Section VI concludes the findings.

II. Related work

A. Software fault prediction

Detecting faulty modules in projects has great significance in software engineering industry. Given a source code, the task is determining whether the code contains bugs (faulty) or not (clean). Existing bugs in software components is unavoidable due to numerous causes like incorrect design, programmers' skills, and the deployment environment. According to practical statistics, fixing bugs after deployment very take time, cost money, and affect to the company reputation. Software fault prediction systems are useful tools for the software industry.

978-1-7281-3003-3/19/$31.00 ©2019 IEEE

There are two main approaches for SFP problems: (1) applying common learning algorithms on handcrafted features called software metrics, and (2) automatically learning program features by deep neural networks. For the first approach, different software metrics have been designed to estimate the relevance of programs to bugs [7]. Other studies made new metrics by combining the available ones [6]. Based on a set of metrics, different machine learning algorithms are applied to build predictive models such as k-nearest neighbors, support vector machines, and neural networks [5]. However, the low quality of software metrics hinder this approach from reaching high accuracy. According to previous studies and analysis, most of current software metrics are statistical measures and they are not highly relevant to the causes of semantic bugs. Recently, researchers have focused on developing deep neural networks to learn program features automatically. In [8], authors used a tree-structured network to classify programs based on abstract syntax trees. In [10], authors applied convolutional neural networks on assembly instruction sequences obtained by compiling source files.

B. Within-project prediction and Cross-project prediction

Preparing data and training models for predicting software faults can be performed by two ways namely within-project and cross-project. Within-project is using the historical or partial data of a project to train predictive models. In the scenario of partial data, project modules are separated into training and test sets, or k parts to conduct $k-$ fold cross validation [5]. However, the lack of data leads to the low performance of this method. One solution is using historical data in which the model is pretrained on previous versions of the project to predict the bugs for the new version. The issue is that many data samples may appear in both training and test sets because some modules remain unchanged.

Unlikely, cross-project uses other projects to train the model for the current project [2]. The motivation behind leveraging data from other sources is that common semantic faults of a programming language may appear in any project. For example, the fault of division by zero can happen anywhere having division. Cross-project aims to tackle the data shortage in the case of within-project prediction. However, since the different data distributions, the model may not generalise to some specific bugs in the target project. Considering rounding operators in a statement, the operators can be a fault of arithmetic precision or clean depending the use scenario.

C. Transfer learning

Recent studies have shown that transfer learning brings deep neural networks closer to practical applications. The key of the success is solving the major challenge of data shortage by reusing the model trained on other tasks. Deep learning with pretrained models has made breakthroughs in many areas including image, natural language, and speech processing [1].

The process of transfer learning is illustrated in the Fig. 1. Instead of training from scratch, we first develop the model on a source task that has numerous data and reuse the model for the target task. Two transfer learning scenarios include 1) fixed feature extractor and 2) fine-tuning. In the first scenario, the final layer is removed and the remainder of the model is used as a feature extractor for the new dataset. After having features, we can build any classifier like softmax or support vector machines. This scenario is suitable for new datasets that are small and similar to the source. For large datasets, the second scenario is commonly adopted. We continue training the whole or some layers of the model from the pretrained.

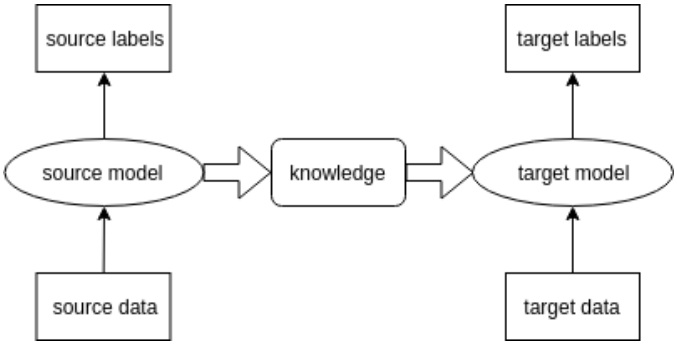

Fig. 1. Learning process of transfer learning.

III. Deep Transfer Learning for Fault Prediction

This section introduces the method to leverage data among software projects to build classifiers. We first describe the base model, then present two scenarios of transfer learning for within and cross-project fault prediction.

A. Sequence-based Convolutional Neural Networks

Our research predicts faulty software modules based on their assembly instruction sequences. Basically, the faults are relevant to the behavior of the program in a specific case and revealed during the execution process. For example, given `C` statement `x = y + 3` where the data type of `x` and `y` is `int`, a fault occurs when the value of the right expression is out of the `int` range. Due to having a one-to-one relationship with machine code, assembly instructions can reveal program behavior rather than software metrics or other representations like abstract syntax trees (ASTs). For these reasons, assembly instructions have been proved to be beneficial to software fault prediction [10], [12].

To learn software faults from assembly instruction sequences, we use a multi-layer convolutional neural network as shown in Fig. 2. The network includes layers of input, embedding, convolution, pooling and fully-connected. Each input is the assembly instruction sequence of a program obtained by compiling the source code. We use an embedding layer that maps a single instruction into a vector to generate the embedding matrix $N \times d = x_1 \oplus x_2 \oplus ... \oplus x_N$ for the input, where, N is the max sequence length and d is the vector dimension; $x_i, .., x_N$ are real-valued vectors of the instructions and $\oplus$ is the concatenation operation. All sequences are padded to the same length with the longest one.
.

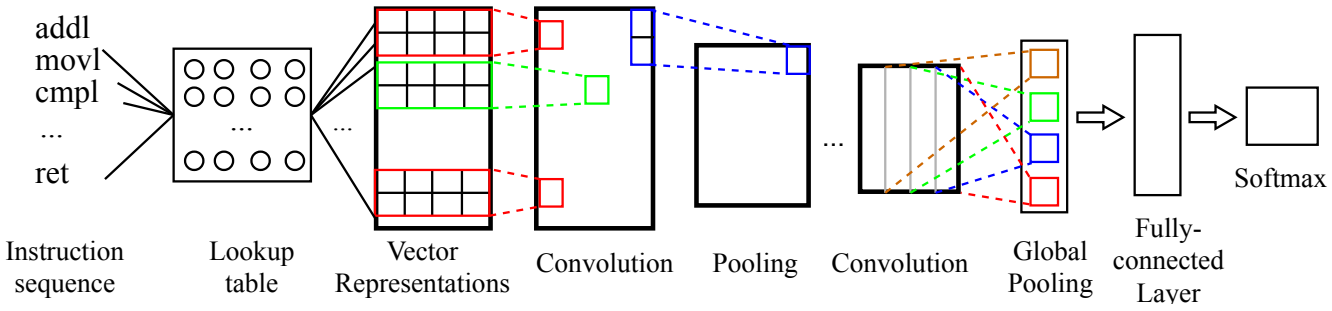

Fig. 2. A sequence-based convolutional neural network for software fault prediction.

Convolutional layers are the key layers of the network. These layers apply a set of filters sliding over the input data to automatically capture the local features at multi-levels of abstraction. For example, a filter with the window size of 2 will learn the relationships of two consecutive instructions. A convolution operation just transforms the data, and remains the length of the input. Formally, given an input sequence with vector representations $X = \{x_1, x_2, ..., x_N\}$, a filter outputs a feature map with the same length $C = \{c_1, c_2, ..., c_N\}$.

$$c_i = f(W \cdot x_{[i:i+h-1]} + b) \quad (1)$$

where $W \in \mathbb{R}^{h \times d}$, $b \in \mathbb{R}$ is the bias, and the function f is the activation.

By adding more convolutional layers, the network can learn more abstract features, and potentially achieve better performance. However, deeper networks are more difficult to train because of vanishing gradient and overfitting problems. Thus, selecting a suitable configuration of convolutional layers is very important.

Pooling layers are to perform dimension reduction of the feature maps and preserve the essential information. As mentioned above, convolution operations remain the input shape. Besides, embedding matrices are high-dimensional since they are the concatenation of thousands of instruction vectors (Table I). For these reasons, reducing the data dimension and the network weights is needed to avoid overfitting. In this scenario, the use of a pooling layer following a convolution is an efficient solution in terms of computational time and implementation.

Downsampling is done separately for each feature map. Some popular pooling functions are max, $average$, and $L2-norm$. Taking max pooling as an example, the function will pick the largest value among considered items. The pooling function applies the operation on sub-regions of the feature map to spatially resize it. As illustrated in Fig. 2, given the window size of 2, the max operation will take max over two numbers (little region 2×1 of the feature map). In summary, the pooling layer will accept the input of $N \times F$ and produce the output of $N/S + 1 \times F$, where N, F, and S are the sequence length, the number of the feature map, and the window size, correspondingly. According to many experiments, max operations have proved better performance than the other ones [3].

In the model, the intermediate pooling layers with a filter size of 2 gradually down the sequence a half. The last pooling layer gathers all information extracted by the convolutional layers to form a feature vector. In this case, we apply the global max-pooling where each feature map is only pooled into a single value.

Fully connected layers are added on the top of the model to build the classifier from the feature vectors learned in the convolution process. We use two fully connected layers, in which the last layer comes with $softmax$ activation to convert the output values into distribution probabilities for classification.

B. Transfer learning for within-project prediction

A highlight of deep neural networks is that they can automatically learn high-quality features. The project modules are divided into training, validation, and test sets. After training the network using the training and validation sets, we remove fully connected layers and select the rest as the feature extractor. The feature extractor then is applied to produce feature vectors for the whole dataset. Finally, we use the features to train support vector machines (SVMs), and k-nearest neighbors (kNN) classifiers for predicting faulty modules.

C. Transfer learning for cross-project prediction

The research question is whether it is possible to leverage programs from different projects to train the classifier. In a programming language, many software modules may have the same issues. For example, the `array index out of range exception`, where we access an item with the index exceeding the array size may appear at any code snippet that manipulates the array data regardless of projects. To verify the research question, we conduct cross-project prediction by training the network on data of other projects and use the model as a feature extractor or fine-tune model on the current project.

IV. Experiments

A. Datasets

We conducted experiments on 4 datasets as in [10] namely SUMTRAIN, FLOW016, MNMX, and SUBINC. Each dataset contains all code submissions to solve a problem on CodeChef

TABLE I
STATISTICS ON FOUR DATASETS ABOUT THE NUMBER OF INSTANCES AND SEQUENCE LENGTH.

Dataset	Total	# Class 0	# Class 1	# Class 2	# Class 3	# Class 4	Max length
FLOW016	10,648	3,472	4,165	231	2,368	412	1,246
MNMX	8,745	5,157	3,073	189	113	213	3073
SUBINC	6,484	3,263	2,685	206	98	232	1,245
SUMTRIAN	21,187	9,132	6,948	419	2,701	1,987	2,095

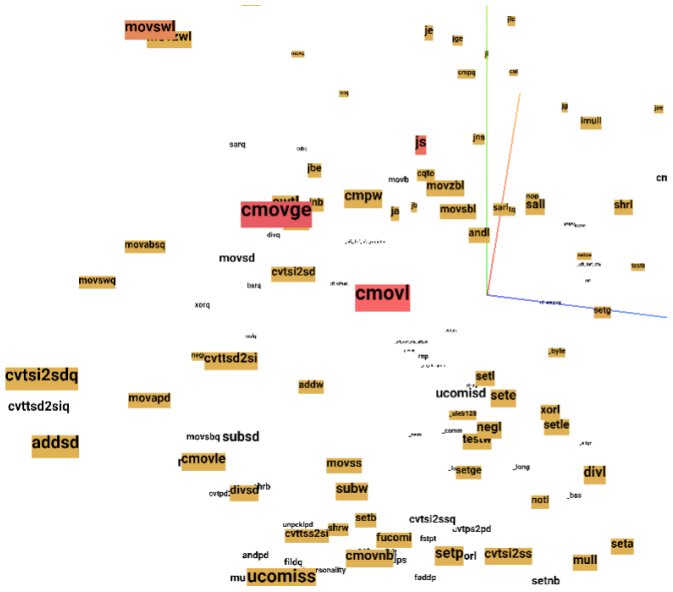

Fig. 3. Visualization of instruction embeddings trained by GLOVE.

[1]. After collecting, the data were preprocessed by removing source files with syntactic errors.

A program is evaluated as one of five outcomes 0) *clean* - the program can run successfully and produce the correct result; 1) *time limit exceeded* - the inefficient implementation that makes program unable to finish the job within the time limit; 2) *the wrong answer* - the output does not match the expected result; 3) *run-time error* - the execution flow is interrupted due to semantic error such as *index out of range* when manipulating array data, or *null-pointer exception* when using a pointer; 4) *syntax error* - the compiler of CodeChef can not compile the program.

Table I shows data statistics on the datasets. As can be seen, all of the datasets are imbalanced and instruction sequences are very long. Considering MNMX dataset, the instance numbers of classes 0 and 1 are much bigger than those of classes 2, 3, 4. Moreover, the instruction sequence length is up to three thousand. Each dataset is randomly split by the ratio 3:1:1 for training, validation, and testing as in [10]. .

B. Instruction embedding

To obtain instruction vector representations, we apply an unsupervised learning algorithm namely GLOVE [9]. GLOVE originally is used to produce word vectors in natural languages.

[1]https://www.codechef.com/problems/⟨problem-name⟩

TABLE II
THE CONFUSION MATRIX

		Actual	
		1	-1
Predicted	1	True Positives (TP)	False Positives (FP)
	-1	False Negatives (FN)	True Negatives (TN)

The embedding model is trained on statistics of global word-word co-occurrences in the corpus.

We collect assembly instructions of programs from all projects to form the corpus and train GLOVE model. The corpus has 174 instructions and the vector dimension is 30. Fig. 3 visualizes the vector representations of assembly instructions. We observe that instructions having similar functions are relevant on the vector space. For example, `cmovl` and *cmovge* are similar since they are data transfer instructions.

C. Hyper parameters

To verify the performance of transfer learning, we compare various approaches including feature-based, and deep neural networks on tree and assembly instruction sequence representations. The model parameters are initialized as follows:

Deep neural networks include some convolution structures based on assembly instructions (ASCNN) [10], abstract syntax trees (TBCNN) [8] and sibling-subtree (SibStCNN) [11], and recursive neural networks (RvNN) [13]. These networks share common hyperparameters including $tanh$ activation function, the initial learning rate of 0.1, instruction embedding dimension of 30.

k-nearest neighbors (kNN) The distance metrics for kNN are tree edit distance (TED) for abstract syntax trees (ASTs), Levenshtein distance (LD) for AST token sequences, and the Euclidean distance for feature vectors extracted by the transfer learning methods. The number of nearest neighbors k is 3.

Support Vector Machines (SVMs) Two SVM classifiers with RBF kernel are applied on bag-of-words (BoW) that counts the instruction occurrences, and the features extracted from the CNN. Two parameters C and γ are selected as 1 and 0, respectively

D. Evaluation Measures

Two measures of accuracy and F1 are used to evaluate the models' performance in which F1 is preferred since the datasets are imbalanced. Two such measures can be computed based on the confusion matrix. Considering the case of binary classification in which samples are grouped into one of two classes $+1$ (positive) or -1 (negative). The accuracy and F1 are calculated by the formulas below:

$$Accuracy = \frac{TP + TN}{TP + FN + FP + TN} \tag{2}$$

$$F1 = 2 \times \frac{precision \times recall}{precision + recall} \tag{3}$$

where $recall = \frac{TP}{TP+FN}$ and $precision = \frac{TP}{TP+FP}$.

For multiclass classification, there are some methods to compute the F1. Since the datasets are imbalanced, weighted

TABLE III
PERFORMANCE COMPARISON AMONG APPROACHES ACCORDING TO ACCURACY AND F1. THE SUPERSCRIPT c AND w INDICATE THE CASES OF CROSS AND WITHIN PREDICTION.

Approach	FLOW016		MNMX		SUBINC		SUMTRIAN	
	Acc.	F1	Acc.	F1	Acc.	F1	Acc.	F1
SVM-BoW	60.00	58.64	77.53	75.00	67.23	65.75	64.87	63.82
kNN-LD	60.75	60.61	79.13	77.89	66.62	66.36	65.81	65.73
kNN-TED	61.69	61.56	80.73	79.55	68.31	68.03	66.97	66.83
RvNN	61.03	58.98	82.56	80.48	64.53	62.07	58.82	56.29
TBCNN	63.10	61.85	82.45	80.94	63.99	62.13	65.05	63.35
SibStCNN	62.25	61.15	82.85	81.04	67.69	65.15	65.10	63.20
ASCNN	72.11	71.36	**83.53**	**82.18**	**73.79**	**73.08**	69.07	69.04
CNN	*74.13**	*73.20**	*80.79**	*79.97**	*69.24**	*68.89**	*66.75**	*66.82**
Transfer-SVM^c	73.71	72.63	82.85	81.28	72.47	72.00	69.89	69.22
Transfer-kNN^c	69.15	68.14	80.50	78.81	70.39	68.89	67.93	67.16
Transfer-SVM^w	**75.35**	**74.37**	83.13	81.85	72.86	72.47	**70.93**	**70.45**
Transfer-kNN^w	72.91	71.79	81.99	80.69	72.71	72.07	68.95	68.33

F1 should be selected. Firstly, we calculate the metrics for all labels and then take the average weighted according to the number of samples for each label.

V. RESULTS AND DISCUSSION

Table III presents the comparison among the different classifiers according to accuracy and F1. Our CNN model is compatible with assembly instruction sequence-based methods that achieve higher performance than those of feature-based and AST-based.

For within-project prediction, the SVM classifier improves the base CNNs significantly on all the datasets. Specifically, in terms of accuracy, it can improve from 74.13% to 75.35% on FLOW016, from 80.79% to 83.13% on MNMX, from 69.24% to 72.86% on SUBINC, and from 66.75% to 70.93% on SUMTRIAN.

For cross-project prediction, the SVM classifier overcomes the base CNN model on three datasets including MNMX, SUBINC, and SUMTRIAN. On FLOW016, the performance is slightly downgraded. Interestingly, in this case, the network is trained totally by the data from other projects. Taking FLOW016 as an example, we use the data from the rest projects to build the feature extractor for FLOW016. According to the above analysis, we can confirm that software projects share common information and leveraging data among projects is beneficial.

It should be noted that, transfer learning can extract high-quality features. Considering kNN classifier with different features including LD, TED, within-project transfer, and cross-project transfer, we can see that kNN with transfer learning achieve higher performance than kNN with other features. This phenomenon is also similar to the case of SVM.

VI. CONCLUSION

In this paper, we verify the efficiency of transfer learning on two main tasks of software fault prediction. We also adopt an unsupervised learning algorithm and observe the quality of assembly instruction embeddings. Our experimental results confirm that since the programs can share common information, leveraging data among projects is beneficial to software fault prediction.

ACKNOWLEDGEMENTS

This research is funded by the Vietnam National Foundation for Science and Technology Development (NAFOSTED) under grant number 102.05-2018.306.

REFERENCES

[1] John Blitzer, Ryan McDonald, and Fernando Pereira. Domain adaptation with structural correspondence learning. In *Proceedings of the 2006 conference on empirical methods in natural language processing*, pages 120–128. Association for Computational Linguistics, 2006.

[2] Gerardo Canfora, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Multi-objective cross-project defect prediction. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 252–261. IEEE, 2013.

[3] Alexis Conneau, Holger Schwenk, Loïc Barrault, and Yann Lecun. Very deep convolutional networks for natural language processing. *arXiv preprint arXiv:1606.01781*, 2016.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[5] David Gray, David Bowes, Neil Davey, Yi Sun, and Bruce Christianson. Using the support vector machine as a classification method for software defect prediction with static code metrics. In *International Conference on Engineering Applications of Neural Networks*, pages 223–234. Springer, 2009.

[6] Taghi M Khoshgoftaar and Edward B Allen. Logistic regression modeling of software quality. *International Journal of Reliability, Quality and Safety Engineering*, 6(04):303–317, 1999.

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

[8] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[9] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[10] Anh Viet Phan and Minh Le Nguyen. Convolutional neural networks on assembly code for predicting software defects. In *2017 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES)*, pages 37–42. IEEE, 2017.

[11] Anh Viet Phan, Minh Le Nguyen, and Lam Thu Bui. Sibstcnn and tbcnn+ knn-ted: New models over tree structures for source code classification. In *International Conference on Intelligent Data Engineering and Automated Learning*, pages 120–128. Springer, 2017.

[12] Anh Viet Phan, Minh Le Nguyen, Yen Lam Hoang Nguyen, and Lam Thu Bui. Dgcnn: A convolutional neural network over large-scale labeled graphs. *Neural Networks*, 108:533–543, 2018.

[13] Richard Socher, Jeffrey Pennington, Eric H Huang, Andrew Y Ng, and Christopher D Manning. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *Proceedings of the conference on empirical methods in natural language processing*, pages 151–161. Association for Computational Linguistics, 2011.

[14] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.

[15] Xiao Yu, Jin Liu, Zijiang Yang, Xiangyang Jia, Qi Ling, and Sizhe Ye. Learning from imbalanced data for predicting the number of software defects. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 78–89. IEEE, 2017.