Convolutional Neural Networks on Assembly Code for Predicting Software Defects

Anh Viet Phan, Minh Le Nguyen
Area of Intelligent Robotics
Japan Advanced Institute of Information Technology
Nomi city, Japan 923-1211
Email: {anhphanviet,nguyenml}@jaist.ac.jp

***Abstract*—Software defect prediction is one of the most attractive research topics in the field of software engineering. The task is to predict whether or not a program contains semantic bugs. Previous studies apply conventional machine learning techniques on software metrics, or deep learning on source code's tree representations called abstract syntax trees.**

This paper formulates an approach for software defect prediction, in which source code firstly is compiled into assembly code and then a multi-view convolutional neural network is applied to automatically learn defect features from the assembly instruction sequences. The experimental results on four real-world datasets indicate that exploiting assembly code is beneficial to detecting semantic bugs. Our approach significantly outperforms baselines that are based on software metrics and abstract syntax trees.

I. Introduction

Software defect prediction is one of the hot topics in the field of software engineering and has important applications. The task is to analyze source files to detect potential buggy code. For a large project, manually investigating defects may be time-consuming because the project contains not only a large number of source files but also many logic connections among its components. Therefore, building automatic systems for detecting and localizing bugs is useful for fixing them quickly and efficiently. This leads to reduce the development efforts, and enhance the quality and reliability of software products.

Many approaches have been proposed to construct models for predicting whether or not a source code contains defects. The traditional methods focus on designing features (also called software metrics) of programs [1], [7], [14]. After that, various machine learning algorithms are applied to build predictive models, e.g, neural networks [6], [11], [17], support vector machines (SVMs) [4], and transfer learning [13]. However, metrics-based methods have been faced many challenges because the existing metrics often fail in capturing the semantics of programs [8], [15].

Recently, several studies have been successful in solving program analysis tasks by applying deep neural networks on source code's abstract syntactic structures, called Abstract Syntax Trees (ASTs). Mou et al. presented a convolutional neural network on ASTs for classifying programs according to functionalities [16]. Wang et al. adopted a deep belief network on sequences of ASTs' tokens to learn features for software defect prediction [22]. Kikuchi et. al addressed source code plagiarism detection by measuring the similarities between tree structures [9].

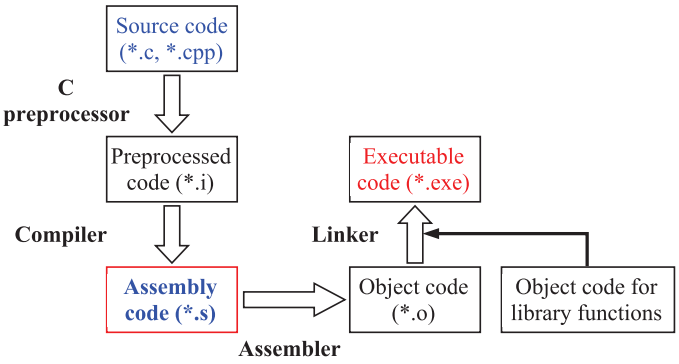

Fig. 1: The process of compiling C programs from source code to executable files.

However, ASTs simply reflect the structures and do not reveal the behavior of programs. Meanwhile, several semantic bugs are hidden deeply in source code and they only are uncovered during running programs in specific cases [23]. For instance, given a C++ statement `z = x/y`, where `x, y, z` are `float`, executing this statement returns unexpected outcomes when `y` equals 0 or values of `x, y and x/y` are out of float range. Therefore, adapting AST-based approaches for software defect prediction may not achieve high performance.

To bridge the gap between programs' semantics and defect features, this paper proposes to employ deep neural networks on sequences of assembly instructions instead of ASTs. Specifically, source files written in C++ language firstly are compiled into assembly code using g++. The predictive models thereafter are built by applying a multiview convolutional neural network to learn on datasets of assembly instruction sequences. Fig. 1 describes the process of compiling C source files into executable files; and Fig. 2 shows the corresponding assembly code of a C program. Assembly files are the products after the compiler analyzes and translates ASTs. These files contain atomic instructions which close to machine code and are executed sequentially. In addition, Convolutional neural networks (CNNs) are known as powerful tools for learning large-scale and high-dimensional datasets [3]. For above reasons, utilizing CNNs and assembly code may be beneficial to learning defect features.

978-1-5386-0743-5/17/$31.00 ©2017 IEEE

This paper, we leverage a convolutional neural network to learn on various information of assembly instruction sequences to build predictive models for software defects. Regarding this, each instruction is viewed by two vectors corresponding to the instruction and its group. Furthermore, all components of an instruction like names, and operands are also embedded to compute its vector. For instance, the vector of instruction `jl L3` is determined based on tokens `jl` and `L3`, and it has an added vector to show that the instruction belongs to "conditional jumps" group. We apply convolution separately to extract features for each view. After that, the extracted features of two views are merged before feeding them to the next layer.

The main contributions of the paper can be summarized as follows:

- Formulate an end-to-end model for software defect prediction by apply multiview convolutional neural networks on assembly instruction sequences.
- Analyze the advantages of utilizing assembly code instead of ASTs, and experimentally prove that exploiting assembly code significantly outperforms AST-based approaches.

The remainder of the paper is organized as follows: Section II explains step-by-step our approach for solving software defect prediction problem from processing data to adapting the learning algorithm. The settings for conducting the experiments such as datasets, algorithm hyper-parameters, and evaluation measures are indicated in Section III. We analyze experimental results in Section III-C, and conclude in Section V.

II. Our Approach

Fig. 3 shows the overall architecture of our model. In the model, each source file is compiled into assembly instruction sequences using g++ on Linux. An instruction can be viewed by some types of information, and each piece of information is represented as a real-valued vector. To automatically learn defect features, we apply different convolutions to discover the underlying meanings of each view. After that, the extracted features are heuristic merged to compute the final scores. In Subsections below, we describe in detail the components in the model.

A. Vector Representations

The first layer of the model is to produce real-valued representations of input sequences. Regarding this, each unique element is assigned a vector in the k-dimensional space, and the values are randomly initialized. After padding to the fixed length of n elements with vectors $x_1, x_2, ..., x_n$, the representation of an input sequence is $x_{1:n} = x_1 \oplus x_2 \oplus ... \oplus x_n$, where $\oplus$ is the concatenation operator.

To enrich the data for training the network, an instruction is viewed by not only the contents (V1) but also the group (V2). For instance, `jne`, `jle`, and `jge` are tagged to the same group since they are conditional jump instructions. Taking V1 into account, the instruction may have many elements including the name and operands. We leverage all of such elements to generate the vector for the V1. Firstly, operands of block names, processor register names, and literal values are replaced by the

```
1  #include <stdio.h>
2  int gcd(int a , int b)
3  {
4   if(b == 0)
5    return a;
6   return gcd(b , a%b);
7  }
8
9  int main(void) {
10  int t;
11  scanf("%d",&t);
12  while(t--)
13  {
14   int a , b;
15   scanf("%d %d", &a ,&b);
16   int  g = gcd(a,b);
17   int l = (a*b)/g;
18   printf("%d %d\n",g,l);
19  }
20  return 0;
21 }
```

(a)

```
1  main:
2  .LFB1:
3      .cfi_startproc
4      pushq   %rbp
5      .cfi_def_cfa_offset 16
6      .cfi_offset 6, -16
7      movq    %rsp, %rbp
8      .cfi_def_cfa_register 6
9      subq    $32, %rsp
10     leaq    -12(%rbp), %rax
11     movq    %rax, %rsi
12     movl    $.LC0, %edi
13     movl    $0, %eax
14     call    scanf
15     jmp .L5
16 .L6:
17     leaq    -20(%rbp), %rdx
18     leaq    -16(%rbp), %rax
19     movq    %rax, %rsi
20     movl    $.LC1, %edi
21     movl    $0, %eax
22     call    scanf
23     movl    -20(%rbp), %edx
```

(b)

Fig. 2: An example of assembly code obtained by compiling a C++ program. (2a) the C source code, (2b) a part of the assembly code.

symbols "name", "reg", and "val", respectively. Following this replacement, the instruction `addq $32, %rsp` is converted into `addq, value, reg`. After that, the vector for V1 of the i^{th} instruction is computed as follows:

$$x_i = \frac{1}{C}\sum_{j=1}^{C} x_j, \tag{1}$$

where C is the number of the components, and x_j is the vector of the j^{th} component.

B. Multiview Convolutional Neural Networks

The network is constructed from four main types of layers including convolutional layer, pooling layer, merge layer and fully-connected layer. Wherein, the convolutional layers are applied to automatically learn defect features from multiple views of instruction sequences. The pooling layers perform down-sampling operations to gather extracted information. The merge layer is to combine feature vectors of all views before feeding to the fully-connected layers (the common layers in regular neural networks) for computing the final class scores.

Convolutional layers play an important role in the success of convolutional neural networks. These layers are efficient to deal with large-scale and high-dimensional data by extracting meaningful statistical patterns. In our model, the latent features of the views are explored independently by applying different convolution operators on the corresponding sequences. We design a set of F feature detectors (filters) to capture local dependencies in the original sequence. Each filter can be viewed as a convolution that slides over the sequence to produce a feature map. Formally, at position i, the feature value of the f^th filter is computed as follows:

$$c_i^f = f(W^f \cdot x_{i:i+h-1} + b^f) \tag{2}$$

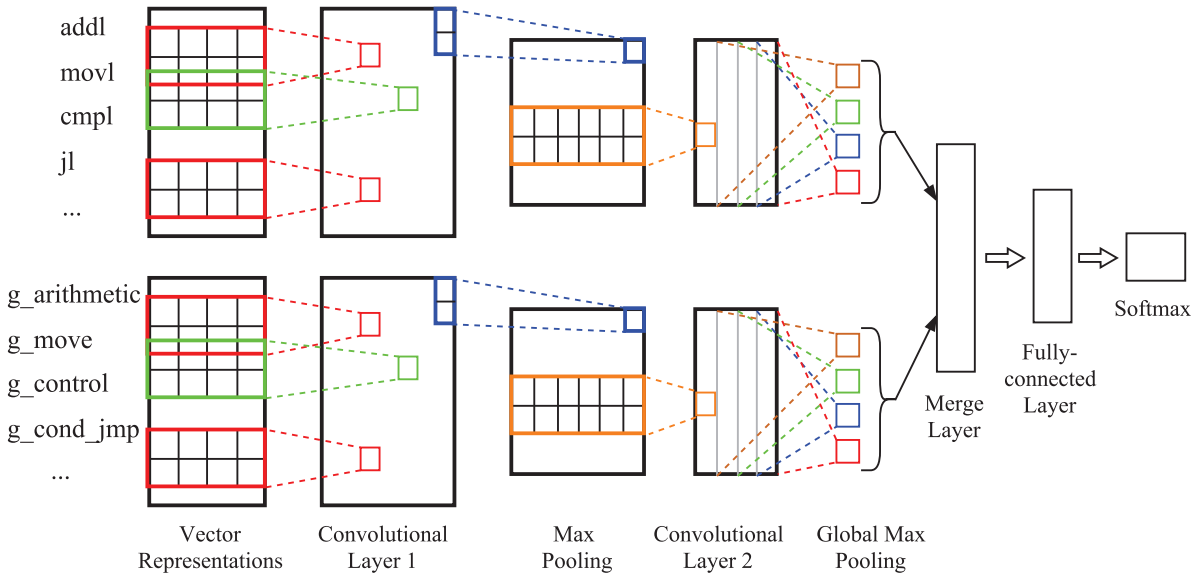

Fig. 3: A convolutional neural network with two views for assembly instruction sequences.

where $W^f \in \mathbb{R}^{h \times k}$, $x_{i:i+h-1} = x_i \oplus x_{i+1} \oplus ... \oplus x_{i+h-1}$, and f is an activation function.

We combine several convolutional layers to make the network able to learn more complex features. In general, deeper networks with multiple stacked convolutional layers potentially achieve better performance [10]. However, using many layers leads to an increase in the number of parameters which must be optimized, and hence a need of large datasets for training networks. In this work, because the datasets are not large, we just stack two layers of convolution for each view.

Pooling layers are commonly inserted between successive convolutional layers is to reduce the dimensions of feature maps but preserve the most important information. In the model, after a convolution, the feature map length is similar to that of the input sequence which has up to thousands of tokens (Table I). Thus, pooling layers are necessary to reduce model parameters, and hence to avoid overfitting.

Downsampling is performed by using a function such as max, average, or sum that takes the largest element, the average, or sum of the input values, respectively. The pooling function is applied on non-overlapping regions to resize the feature map spatially. According to various studies, max pooling has shown better results than other pooling types [2].

In the model, the intermediate convolutions are followed by a local max-pooling layer with the filter size of 2. For the last convolution, a global max-pooling is applied to generate the vector representation for the corresponding view, in which each element is the result of pooling a feature map.

Merge layer

After convolution and the pooling steps, we obtain vector representations for the sequences of all views. Such feature vectors are combined before feeding to the fully-connected layer for estimating the categorical distribution for a program. We examine several merge operations such as concatenation, element-wise multiplication, and element-wise maximization.

III. Datasets and Experimental Setup

A. Datasets

The datasets for conducting experiments are obtained from a popular programming contest site CodeChef [1]. We created four benchmark datasets which each one involves source code submissions (written in C, C++, Python, etc.) for solving one of the problems as follows:

- SUMTRIAN (Sums in a Triangle): Given a lower triangular matrix of n rows, find the longest path among all paths starting from the top towards the base, in which each movement on a part is either directly below or diagonally below to the right. The length of a path is the sums of numbers that appear on that path.
- FLOW016 (GCD and LCM): Find the greatest common divisor (GCD) and the least common multiple (LCM) of each pair of input integers A and B.
- MNMX (Minimum Maximum): Given an array A consisting of N distinct integers, find the minimum sum of cost to convert the array into a single element by following operations: select a pair of adjacent integers and remove the larger one of these two. For each operation, the size of the array is decreased by 1. The cost of this operation will be equal to their smaller.
- SUBINC (Count Subarrays): Given an array A of N elements, count the number of non-decreasing subarrays of array A.

The target label of an instance is one of the possibilities of source code assessment. Regarding this, a program can be assigned to one of the groups as follows: 0) accepted - the program ran successfully and gave a correct answer; 1) time

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

TABLE I: Statistics on the datasets. Max length is the length of the longest assembly instruction sequence.

Dataset	Total	Class 0	Class 1	Class 2	Class 3	Class 4	Max length
FLOW016	10648	3472	4165	231	2368	412	1246
MNMX	8745	5157	3073	189	113	213	3073
SUBINC	6484	3263	2685	206	98	232	1245
SUMTRIAN	21187	9132	6948	419	2701	1987	2095

TABLE II: Structures of the neural networks. Each layer is presented in form of the name followed by the number of neurons or the size of the filters in case of pooling. Emb is an embedding layer. Rv, TC, SC, GPool, and FC stand for recursive, tree-based convolutional, sequence-based convolutional, global pooling, and fully-connected, respectively.

Network	Architecture
RvNN	Coding30-Emb30-Rv600-FC600-Soft5
TBCNN	Coding30-Emb30-TC600-GPool-FC600-Soft5
SibStCNN	Coding30-Emb30-TC600-GPool-FC600-Soft5
ASCNN-1V	SC600-Pool2-SC300-GPool-FC300-Soft5
ASCNN-2V	[SC600-Pool2-SC300-GPool]×2-Merge-FC300-Soft5

limit exceeded - the program was compiled successfully, but it did not stop before the time limit; 2) wrong answer: the program compiled and ran successfully but the output did not match the expected output; 3) runtime error: the code compiled and ran but encountered an error due to reasons such as using too much memory or dividing by zero; 4) syntax error - the code was unable to compile.

We collected all submissions written in C or C++ until March 14th, 2017 of four problems. The data are preprocessed by removing source code which are existing syntax errors, empty code, and the code. Table I presents statistical figures of instances in each class of the datasets. All of the datasets are imbalanced. Taking MNMX dataset as an example, the ratios of classes 2, 3, 4 to class 0 are 1 to 27, 46, and 24. To conduct experiments, each dataset is randomly split into three folds for training, validation, and testing by ratio 3:1:1.

B. Experimental Setup

To verify the efficiency of using assembly instruction sequences, we compare with support vector machines (SVMs) based on hand-crafted features and other deep neural networks based on tree representations. For the tree-based approaches, a source code is converted to an abstract syntax tree (AST) using a parser, and then computational models are employed to discover the latent information of trees. The hyperparameters for the baselines are described as follows:

The neural networks For neural networks including convolutions on assembly instruction sequences (ASCNN), tree-based convolutional neural networks (TBCNN) [16], Sibling-subtree convolutional neural networks (SibStCNN - an extension of TBCNN in which feature detectors are redesigned to cover subtrees including a node, its descendants, and siblings), and recursive neural networks (RvNN) [20], we use the common settings including the activation function is $tanh$, initial learning rate is 0.1, vector size of tokens is 30. The structures of the networks are shown in Table II.

k-nearest neighbors (kNN) We apply kNN algorithm with tree edit distance (TED) and Levenshtein distance (LD) [18]. The number of neighbors k is selected from 3, 5, 7, 9.

Support Vector Machines (SVMs) The SVM classifiers are built based on hand-crafted features, namely bag-of-words (BoW). By this way, the feature vector of each program is determined by counting the numbers of times symbols appear in the AST. The SVM with RBF kernel has two parameters C and γ; their values are 1 and 0, respectively.

C. Evaluation Measures

The approaches are evaluated based on three widely used measures including accuracy, F1, and AUC. For imbalanced datasets, F1 and AUC are more suitable to rank classifiers.

Predictive accuracy is the ratio of correctly classified instances to the total number of instances. The accuracy is one of the most important criteria to assess classification performance.

However, accuracy is not enough to confirm the quality of a classifier, especially in the case of imbalanced data. Imbalanced datasets are that some classes have much more samples than others, most of the standard algorithms are biased towards the major classes and ignore the minor classes. Consequently, the hit rates on minor classes are very low, although the overall accuracy may be high. Meanwhile, in practical applications, accurately predicting minority samples may be more important. Taking account of software defect prediction, the essential task is detecting faulty modules. However, many software defect datasets are highly imbalanced and the faulty instances belong to minority classes [19].

Because all experimental datasets are imbalanced, F1 and AUC should be taken into account. For binary classification that data samples are classified into two categories: positive and negative, F1 is calculated as follows:

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

where precision and recall are the ratios of correct positive instances to all positive instances, and to predicted positive instances, respectively. For multiclass classification, we compute the metrics for each label and F1 is the weighted average based on the number of instances for each label.

The area under the receiver operating characteristic (ROC) curve, known as AUC, estimates the discrimination ability between classes is an important measure to judge the effectiveness of algorithms. It is equivalent to the non-parametric Wilcoxon test in ranking classifiers [5]. According to previous research, AUC has been proved as a better and more statistically consistent criterion than the accuracy [12], especially for imbalanced data.

ROC curves which depict the tradeoffs between hit rates and false alarm rates are commonly used for analyzing binary classifiers. To extend the use of ROC curves to multi-class problems, the average results are computed based on two

TABLE III: Comparison of classifiers according to accuracy, F1, and AUC measures. 1V and 2V following ASCNN means that an instruction are viewed by one and two perspectives. Op and NoOp are using instructions with or without operands.

Approach	FLOW016			MNMX			SUBINC			SUMTRIAN		
	Acc.	F1	AUC	Acc.	F1	AUC	Acc.	F1	AUC	Acc.	F1	AUC
SVM-BoW	60.00	58.64	0.74	77.53	75.00	0.76	67.23	65.75	0.73	64.87	63.82	0.79
LD	60.75	60.61	-	79.13	77.89	-	66.62	66.36	-	65.81	65.73	-
TED	61.69	61.56	-	80.73	79.55	-	68.31	68.03	-	66.97	66.83	-
RvNN	61.03	58.98	0.75	82.56	80.48	0.79	64.53	62.07	0.69	58.82	56.29	0.73
TBCNN	63.10	61.85	0.76	82.45	80.94	0.77	63.99	62.13	0.72	65.05	63.35	0.78
SibStCNN	62.25	61.15	0.76	82.85	81.04	0.79	67.69	65.15	0.71	65.10	63.20	0.80
ASCNN_1V_NoOp	72.11	71.36	0.81	83.53	82.18	0.80	**73.79**	**73.08**	0.72	**69.07**	**69.04**	0.80
ASCNN_2V_NoOp	**73.47**	**72.79**	0.81	82.96	81.67	0.79	73.71	73.04	**0.75**	67.82	67.81	**0.81**
ASCNN_1V_Op	72.39	71.57	0.82	**84.33**	**82.67**	0.80	72.01	71.54	**0.75**	66.52	67.12	**0.81**
ASCNN_2V_Op	73.19	72.71	**0.83**	82.73	81.14	**0.82**	73.63	72.88	**0.75**	68.29	68.19	0.80

TABLE IV: The performance of model variants

Model variant		FLOW016			MNMX			SUBINC			SUMTRIAN		
	Merge	Acc.	F1	AUC	Acc.	F1	AUC	Acc.	F1	AUC	Acc.	F1	AUC
NoOp	concatenate	**73.47**	**72.79**	*0.81*	82.96	81.67	*0.79*	73.71	73.04	**0.75**	*67.82*	*67.81*	0.81
	maximum	*71.78*	*71.30*	**0.82**	82.73	81.75	*0.79*	**73.86**	**73.35**	0.74	**69.89**	**69.56**	0.81
	average	73.38	72.47	**0.82**	**84.39**	**82.96**	**0.81**	73.01	72.28	0.74	69.40	68.71	0.81
	add	72.58	72.29	*0.81*	82.96	81.72	**0.81**	72.94	72.22	*0.73*	69.35	69.08	0.81
	multiply	72.82	71.94	*0.81*	*82.16*	*80.59*	**0.81**	*71.16*	*70.55*	**0.75**	69.21	68.91	0.81
Op	concatenate	73.19	72.71	**0.83**	82.73	81.14	**0.82**	73.63	72.88	0.75	*68.29*	68.19	*0.80*
	maximum	73.38	72.92	*0.82*	**82.90**	**81.16**	**0.82**	**73.94**	**73.06**	**0.76**	69.40	68.89	*0.80*
	average	73.00	72.18	**0.83**	82.45	80.94	0.79	71.78	71.21	0.75	70.65	70.29	**0.82**
	add	**73.43**	**73.03**	*0.82*	82.39	80.42	0.79	73.79	73.01	0.75	**71.59**	**71.20**	**0.82**
	multiply	*71.64*	*70.67*	*0.82*	*79.13*	*76.29*	*0.78*	*52.51*	*36.15*	*0.56*	68.83	*67.76*	0.81

ways: 1) macro-averaging gives equal weight to each class, and 2) micro-averaging gives equal weight to the decision of each sample [21]. The AUC measure for ranking classifiers is estimated by the area under the macro-averaged ROC curves.

IV. Results and Discussion

Table III compares the performance of classifiers on four experimental datasets. For two-view ASCNN, the merge layer uses concatenation operation for combining two view vectors. As can be seen, ASCNN outperforms other approaches according to all measures. This means that learning from assembly code not only improves the accuracy but also increases the discrimination ability between classes. Specifically, ASCNN achieves higher accuracies than the second best 10.37% on FLOW016, 1.48% on MNMX, 5.48% on SUBINC, and 2.1% on SUMTRIAN. Similarly, ASCNN enhances F1 values by 10.94% on FLOW016, 1.63% on MNMX, 5.05% on SUBINC, and 2.21% on SUMTRIAN. After analyzing the data we found that ASTs just reflect the structures of programs. Meanwhile, software defect prediction is a complicated task because semantic errors are hidden deeply in source code. Even if a defect exists in a program, it is only revealed during running the application under specific conditions. As a result, although tree-based approaches (SibStCNN, TBCNN, and RvNN) are successfully applied to other software engineering tasks like classifying programs by functionalities, they have not shown good performance on the software defect prediction. In contrast, assembly code contains atomic instructions that are close to machine code. Therefore, learning from assembly code may be beneficial to detecting semantic bugs.

We also analyze the effects of taking advantages of varying information to the classification performance. From the last four rows of Table III, providing more data is beneficial to the model according to AUC. Indeed, both adding group information and considering instruction components boost the ability to distinguish between instances of classes. Especially in the case of ASCNN_2V_Op, it usually achieves highest AUCs. It should be noted that embedding operand vectors in the instruction vector leads to decreases in accuracy and F1. On the average, the ASCNN models without operands (NoOp) obtain higher accuracies and F1 scores than those with operands (Op). The problem is probably because the number of unique tokens increases notably while the datasets are not so large. Specifically, the number of tokens is 174 in the case of NoOp, and is 360 in the case of Op.

For two-view ASCNN, the merge layer is used to combine vector representations of individual sequences to form the feature vector of the program. The combination operation determines the amount of information that the representations are absorbed and hence affects the quality of the feature vector. We survey several $O(1)$ merge operations due to the computational time efficiency including concatenation, and several element-wise operators: addition, average, maximization, and multiplication. Table IV shows the performance of ASCNN with different merge operations. In general, the operations reach high results except for multiplication. The multiplication usually obtains the worst accuracy, F1, and accuracy. Especially on the SUBINC dataset, the ASCNN's performance is degraded drastically.

V. Conclusion

In this paper, we presented an approach for predicting software defects in source code by applying a multiview convolutional neural network to learn from assembly code. Because semantic bugs usually expose via program behavior and assembly code is close to machine code, exploiting assembly instruction sequences is more useful than ASTs that simply reflect the structures of programs. Our experiments indicate that due to learning from the explicit representation and various information of programs, the proposed model significantly enhances the performance of feature-based and tree-based approaches.

Acknowledgements

This work was supported partly by JSPS KAKENHI Grant number 15K16048 and the first author would like to thank the scholarship from Ministry of Training and Education (MOET), Vietnam under the project 911.

References

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

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

[3] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pages 3844–3852, 2016.

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

[5] Tom Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[6] Iker Gondra. Applying machine learning to software fault-proneness prediction. *Journal of Systems and Software*, 81(2):186–195, 2008.

[7] Maurice Howard Halstead. *Elements of software science*, volume 7. Elsevier New York, 1977.

[8] C Jones. Strengths and weaknesses of software metrics. *AMERICAN PROGRAMMER*, 10:44–49, 1997.

[9] Hiroshi Kikuchi, Takaaki Goto, Mitsuo Wakatsuki, and Tetsuro Nishino. A source code plagiarism detecting method using alignment with abstract syntax tree elements. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2014 15th IEEE/ACIS International Conference on*, pages 1–6. IEEE, 2014.

[10] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[11] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.

[12] Charles X Ling, Jin Huang, and Harry Zhang. Auc: a statistically consistent and more discriminating measure than accuracy. In *IJCAI*, volume 3, pages 519–524, 2003.

[13] Ying Ma, Guangchun Luo, Xue Zeng, and Aiguo Chen. Transfer learning for cross-company software defect prediction. *Information and Software Technology*, 54(3):248–256, 2012.

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

[15] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33(1), 2007.

[16] 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.

[17] Ahmet Okutan and Olcay Taner Yıldız. Software defect prediction using bayesian networks. *Empirical Software Engineering*, 19(1):154–181, 2014.

[18] Viet Anh Phan, Ngoc Phuong Chau, and Minh Le Nguyen. Exploiting tree structures for classifying programs by functionalities. In *Knowledge and Systems Engineering (KSE), 2016 Eighth International Conference on*, pages 85–90. IEEE, 2016.

[19] Daniel Rodriguez, Israel Herraiz, Rachel Harrison, Javier Dolado, and José C Riquelme. Preliminary comparison of techniques for dealing with imbalance in software defect prediction. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, page 43. ACM, 2014.

[20] 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.

[21] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 45(4):427–437, 2009.

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

[23] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 334–345. IEEE, 2015.