APPLYING DEEP LEARNING ON TREE AND GRAPH STRUCTURES FOR PROGRAM ANALYSIS

PHAN, ANH VIET

Japan Advanced Institute of Science and Technology

Doctoral Dissertation

APPLYING DEEP LEARNING ON TREE AND GRAPH STRUCTURES FOR PROGRAM ANALYSIS

PHAN, ANH VIET

Supervisor: NGUYEN Minh Le

School of Information Science
Japan Advanced Institute of Science and Technology

March 2018

Abstract

The rapid growth of software industry has increased a high demand for tools based on source code analysis to support developers and managers during software development. Source code classifiers are used to organize big projects or a huge amount of open source code on the web, and thus facilitate software reuse and maintenance. With a software defect prediction tool, programmers can easily locate and fix bugs. This leads to an increase in the software quality, and a decrease in the development time and product cost.

Solving software engineering problems is a big challenge. According to previous studies, programming languages contain abundant statistical properties that are difficult to capture by humans. In addition, a program may show different actions in different cases hindering us from discovering its semantic meaning. Computers can run programs by just executing single instructions, they do not truly understand the programs. For these reasons, although many efforts have made to solve software engineering problems, the achievements are not so high. The traditional approaches build predictive models based on machine learning algorithms and handcrafted features, called software metrics. The drawbacks of such approaches are time-consuming and inaccurate because we must to manually design a set of appropriate metrics and the existing metrics are not enough to capture semantic meanings of programs. Recently, applying deep learning on tree representations to automatically learn programs' features has made a breakthrough in source code analysis. However, such trees simply reflect the program structures and do not reveal the behavior of programs. Thus, tree-based approaches may be inefficient when adapting to several tasks, especially those are relevant to an understanding of semantic meanings like software defect prediction.

In this dissertation, we focus on two main tasks: (1) proposing models and techniques to enhance existing approaches, and (2) formulating a new approach program analysis. For software metrics-based methods, we design a feature weighting model to estimate the importance extent of each metric according to its relevance to class labels. For tree-based approaches, we develop new models as well as refine data by pruning redundant branches to boost the performance. Additionally, we propose a new approach that applies deep learning on assembly code to explore deeper into semantic meanings of programs.

Our contributions can boost the performance of current methods notably and be adapted to various problems of source code analysis.

Keywords: Program Analysis, Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), Deep Learning, Convolutional Neural Networks (CNNs).

Acknowledgments

This dissertation would not have been accomplished without the support of many people. First and foremost, I would like to express my sincere thank to my great supervisor, Professor Nguyen Minh Le at School of Information Science of JAIST for the dedicated guidance through my study time. Through many meetings, he inspired me with the research enthusiasm and pushed me into significant tasks. When I expressed my ideas he patiently listened and uncovered the more fundamental story. I am very grateful to Nguyen Minh for teaching me not only the way to formulate research ideas but also the way to think. I am really happy and so proud to be one of his students.

I would like to express my special thanks to Professor Mizuhito Ogawa, who guided me to do the sub-theme. He provided many background knowledge and encouraged me to understand deeper neural networks and deep learning. When I prepared a terrible slides drafts, he patiently arranged several-hour discussions until all contents were clear. Thanks to the discussions with him, I discovered a new method to solve my research problems.

I would like to thank committee members consisting of Professor Satoshi Tojo, Professor Mizuhito Ogawa, Associate Professor Okada Shogo, and Professor Tomoko Matsui who have spent their valuable time for reading my thesis and giving me constructive comments. My thesis is improved very much thanks to their comments.

I am thankful to several organizations for funding my research including JSPS KAKENHI Grant and Vietnam's Ministry of Education and Training (MOET). Their financial support gave me chances to study in Japan and attend international conferences.

I have to thank many members in Machine Learning and Language Understanding Laboratory, who have contributed to my daily life at JAIST, especially, Chien Tran, Vu Tran and Viet Lai for setting up the servers for running experiments, Tien Nguyen and Danilo for their valuable comments on my work.

I would like to sincerely thank professors in the English Language Education for Science, Technology, and Engineering center of JAIST for reading and editing my manuscripts, and all JAST staff for their kind help in completing necessary procedures.

Deep in my heart, I would like to give special thanks to my family. My parents At Phan and Nhuan Le, and my siblings Tai Phan, Tu Phan and Hai Phan, who always believe me than I do. Kindness, hard work and the passion for exploration are that I learned from you. My wife and little son, your sacrifice, love, and encouragement motivated me to overcome any hindrance.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Program Analysis Problems

Solving software engineering (SE) problems has great significance not only in research but also in practice due to advantages it brings to the software industry. For instance, accurately predicting defects in software modules in early stages helps to direct test efforts, reduce costs, improve the software quality and satisfy customers [90]. Estimating software maintenance efforts supports managers in running projects more efficiently via timely and reasonable adjustment of resources and staff [2]. Source code classification accordingly functionalities is beneficial to software reuse and big project organization. Thus, increasingly researchers are showing interest in applying machine learning techniques to tackle SE problems. My research mainly focuses on solving two tasks including software defect prediction and source code classification.

Software defect prediction is the task to predict whether or not a code contains defects. A defect is an error or a bug in a computer program that causes incorrect or unexpected results when executing the program. A program with some defects is called a buggy code, otherwise, is a clean code. Software defects are hidden deeply in source code and only revealed in specific conditions. Additionally, some defects are ambiguous because depending on the purpose of the design, a code can be considered as a buggy code or a clean code. Let consider two `C` code snippets below.

Program 1

```c
int sumtoN(int N)
{
   int sum=0;
   for(int i=1; i<=N;)
   {
      sum += i;
      i=1;
   }
   return sum;
}
```

Program 2

```c
int main()
{
   int a, b, c;
   printf("Please input two numbers:");
   scanf("%d",&a);
   scanf("%d",&b);
   c = a/b;
   printf("%d / %d = %d", a, b, c);
   return 0;
}
```

Program 1 has a potential defect of an infinite loop. Indeed, after each iteration, the control variable i is reassigned to 1, if the value of N is greater or equal 1, the condition `i<=n` is always satisfied. The `for` statement is repeated endlessly. Similarly, there are several types of defects in program 2. The first is the arithmetic overflow or underflow. Since the program allows to input two integer values without checking, the users may put a value that is outside of the integer range. The second is the division by zero occurring when `b` equals 0. Possibly there is a defect of the loss of arithmetic precision due to rounding. Assuming that `a` and `b` have the values of 3 and 2, the value of `c` after computing the division is 1. However, in certain situations, the desired value should be 1.5.

In practice, software defects may lead to serious consequences such as the loss of money, time, business credibility, and even loss of life. A noticeable example is the case of the Therac-25, a radiation therapy machine for treating cancer patients in 1980s. Bugs in the control program made the machine produced radiation doses with hundreds of times greater than the expectation, and were the cause of several deaths and serious injuries. In 1996, because of specification and design errors, an Ariane 5 rocket of the European Space Agency burst several minutes after launched. The estimated cost for a decade of development and the value of the rocket and its cargo is about 7.5 billion dollars. Recently, a study estimated that every year software bugs cost the US economy 59.5 billion dollars, and more than a third of such amount can be eliminated by improving testing.

Validating and verifying software products before deploying are essential. Existing defects in software components is unavoidable due to various reasons like human mistakes. As above analysis, defects only reveal in specific conditions, although many efforts, testers could not discover all issues in products. Therefore, building a tool for automatically predicting defects is beneficial to software development. With this tool, developers will pay more attention to potentially buggy components to localize and fix defects. This helps to enhance software quality, reduce the development time and efforts, satisfy customers.

Source code classification is the task to tag programs into different categories according to criteria such as application categories and programming languages. Properly organizing software repositories is beneficial to programmer cooperation, maintenance and reuse of software. Within a project, grouping source files according to their functions helps developers can easily locate defect locations when receiving the test reports. Nowadays, programming communities have contributed a huge amount of source code to the internet.

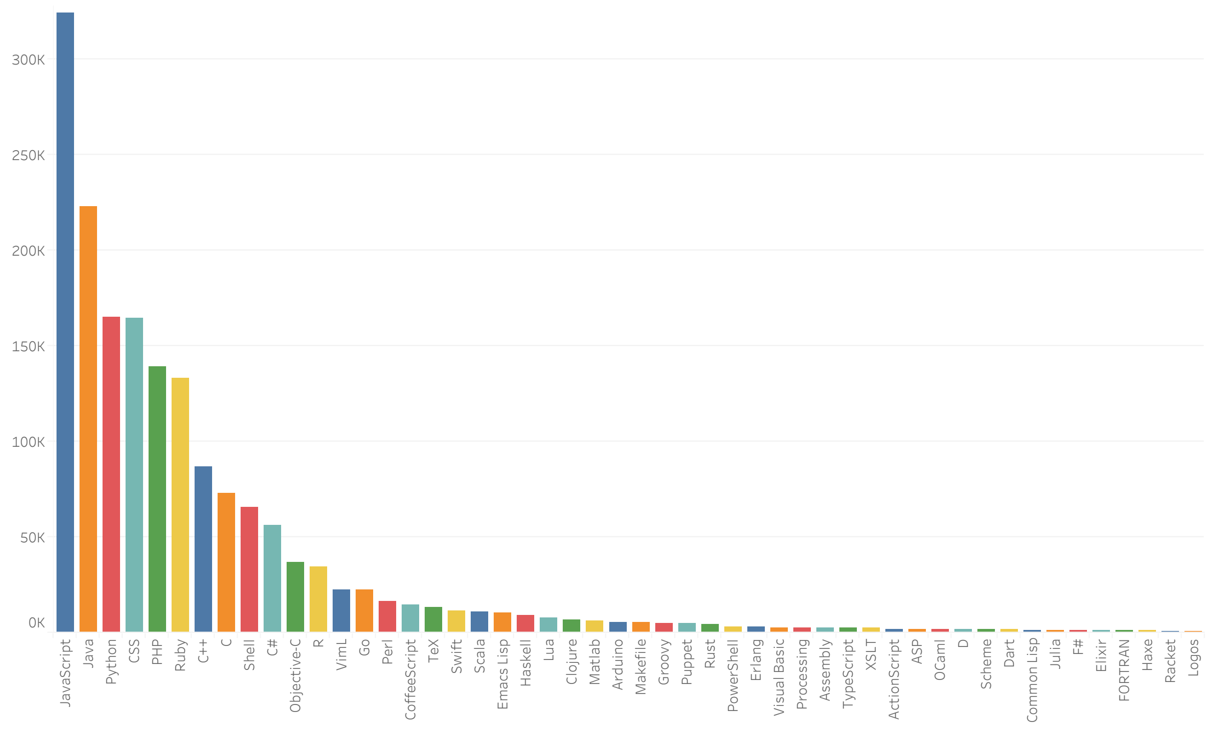


Figure 1.1: Active repositories of the most used languages on GitHub.

Furthermore, this code can be deployed widely and verified by many users. They are valuable resources that we can reuse to develop new systems with less time and efforts than building from scratch. Fig. 1.1 shows active repositories (having at least one code push during a period) of the most used languages on GitHub during the final quarter of 2014. Without an appropriate organization, it is difficult to find a source code compatible with our demands. Manually pushing a code into a suitable group is impractical because sizes of projects are very large and increase rapidly [88]. A source code classifier is a tool to support this work efficiently.

1.2 Conventional Approaches and Limitations

The traditional approaches can be divided into two directions that are based on software metrics and abstract syntax trees. For metrics-based methods, we must manually design a set of features called software metrics to measure some properties of source code. Then various common machine learning algorithms are investigated to build predictive models on the metric data. For tree-based methods, each source code written in a programming language with grammar is represented as a tree called abstract syntax tree (AST). Fig. 1.2 illustrates the AST of the `C` statement "`printf("The sum of x+y =%d", x+y);`". Deep neural networks thereafter are applied to automatically learn distinguishing features from ASTs. With such features, we can build models based on either the deep neural networks directly or other learning algorithms.

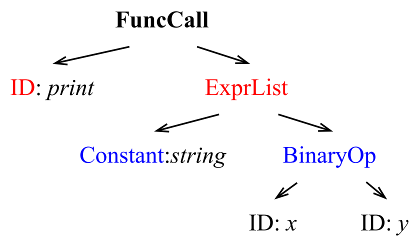


Figure 1.2: The AST of the statement "`printf("The sum of x+y =%d", x+y);`".

The metrics-based approaches are also widely applied to solve various software engineering problems such as fault prediction, cost, and effort estimation [38,45,53,91]. However, the meaning of software metric values have been widely debated for two major reasons: they have not shown good ability to capture the underlying meaning of programs [62]; and most of the currently used metrics have multiple definitions and ambiguous counting rules [40]. Akiyama et al. [1] predicted defects from lines of code (LOC) by Eq.(1.1).

$$\#of\ defects = 4.86 + 0.018 * LOC \tag{1.1}$$

Halstead et al. [35] proposed several complexity metrics and used these as predictors of program defects. The most notable predictor asserted by the author is computed based on numbers of unique operators and unique operands as follows.

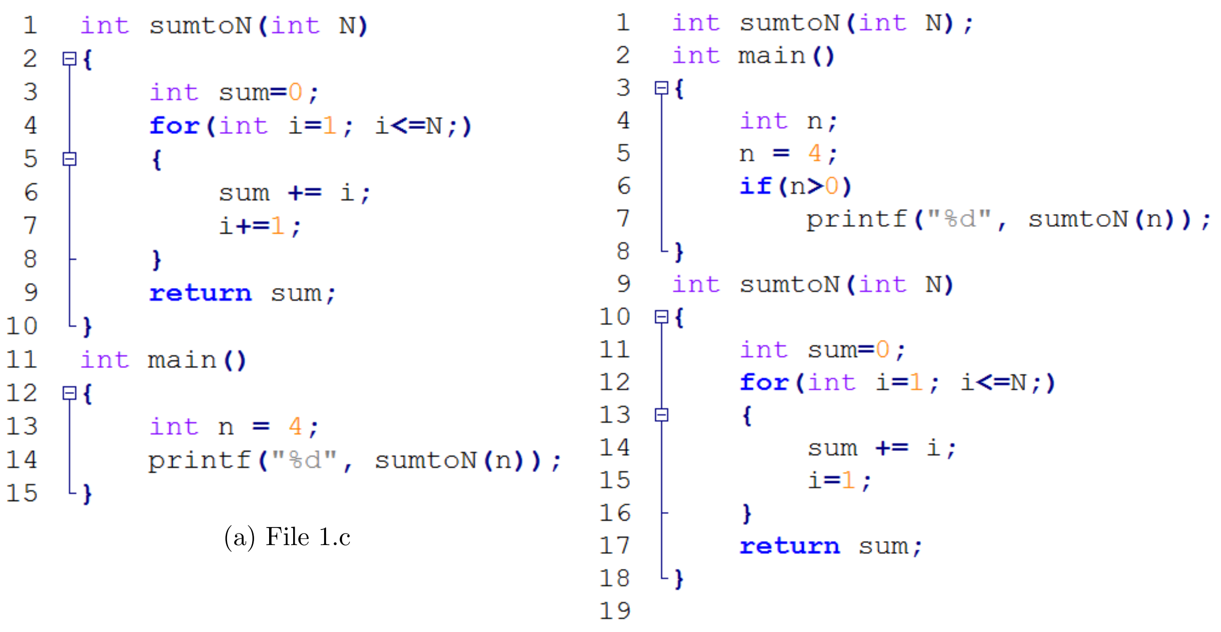
$$\#of\ defects = volume/3000 \tag{1.2}$$

where $volume = N * log2n$, n is the number of operators and operands in the program.

In fact, the defect rates are relevant to programmers' skills, the complexity of projects and other factors rather than the LOC, number of operands or operators. Hence although various robust machine learning algorithms have been applied, the predictors have not achieved so high performance. According to recent studies, the mean probability of detection (PD) on NASA MDP datasets [78] is around 71% [13,62].

Recently, several software engineering problems have been successfully solved by exploiting ASTs of programs [58]. In the field of machine learning, the quality of input data directly affects the performance of learners. Regarding this, due to containing rich information of programs, tree-based approaches have shown significant improvements in comparison with previous research based on software metrics. Mou et al. proposed a tree-based convolutional neural network to extract structural information of ASTs for classifying programs by functionalities [66]. Wang et al. employed a deep belief network to automatically learn semantic features from AST tokens for defect prediction [92]. Kikuchi et. al measured the similarities between tree structures for source code plagiarism detection [47].

Although AST-based approaches have made a breakthrough in source code classification, they may fail to tackle software defect prediction. Defect characteristics are deeply hidden in programs' semantics and they only cause unexpected output in specific conditions [93]. Meanwhile, ASTs do not show the execution process of programs; instead, they



```
 1  int sumtoN(int N)
 2  {
 3      int sum=0;
 4      for(int i=1; i<=N;)
 5      {
 6          sum += i;
 7          i+=1;
 8      }
 9      return sum;
10  }
11  int main()
12  {
13      int n = 4;
14      printf("%d", sumtoN(n));
15  }
```

(a) File 1.c

```
 1  int sumtoN(int N);
 2  int main()
 3  {
 4      int n;
 5      n = 4;
 6      if(n>0)
 7          printf("%d", sumtoN(n));
 8  }
 9  int sumtoN(int N)
10  {
11      int sum=0;
12      for(int i=1; i<=N;)
13      {
14          sum += i;
15          i=1;
16      }
17      return sum;
18  }
19
```

(b) File 2.c

Figure 1.3: A motivating example

simply represent the abstract syntactic structure of source code. Therefore, both software metrics and AST features may not reveal many types of defects in programs. For example, we consider the procedures with the same name `sumtoN` in two C files `File 1.c` and `File 2.c` (Fig. 1.3). Two procedures have a tiny difference at line 7 `File 1.c` and line 15 in `File 2.c`. As can be seen a bug from `File 2.c`, the statement `i=1` causes an infinite loop of the `for` statement in case `N>=1`. Whereas using RSM tool[1] to extract the traditional metrics, their feature vectors are exactly matching, since two procedures have the same lines of code, programming tokens, etc. Similarly, parsing the procedures into ASTs using Pycparser[2], their ASTs are identical. These mean that both metrics-based and tree-based approaches are not able to distinguish the two programs.

1.3 Motivation and Contributions

In the field of software engineering, exploring structures and semantic meanings of programs helps us solve various practical problems including program classification [66, 88], software defect prediction [53, 60], software plagiarism [14, 56] and malware analysis [5, 12]. In this thesis, I focus on addressing the drawbacks of existing methods to boost the performance, and formulating a new approach for program analysis.

For metrics-based approaches, most of the current studies treat all metrics with the same role. They concentrate on finding suitable learning algorithms [31, 97], pre-processing

[1]http://msquaredtechnologies.com/m2rsm/
[2]https://pypi.python.org/pypi/pycparser

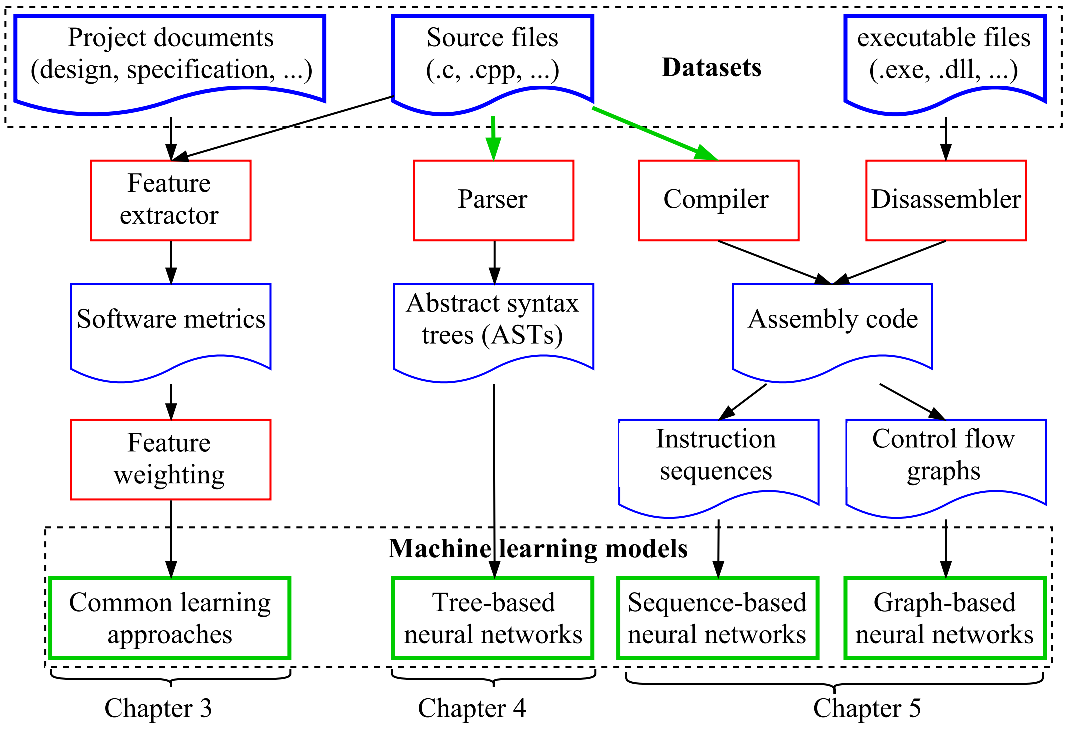

Project documents (design, specification, ...)
Source files (.c, .cpp, ...)
Datasets
executable files (.exe, .dll, ...)
Feature extractor
Parser
Compiler
Disassembler
Software metrics
Abstract syntax trees (ASTs)
Assembly code
Feature weighting
Instruction sequences
Control flow graphs
Machine learning models
Common learning approaches
Tree-based neural networks
Sequence-based neural networks
Graph-based neural networks
Chapter 3
Chapter 4
Chapter 5

Figure 1.4: Illustration of our work in this thesis.

data to remove duplicated and inconsistent instances [75], and selecting the best set of features [46]. In fact, depending on each problem, the extent of the relevance between metrics and class labels is different. For example, the defect rates are relevant to programmers' skills, the complexity of projects and other factors rather than the lines of code, the number of operands or operators. Thus, measures that reflect the software complexity like Halstead complexity and McCabe's complexity should be emphasized. To do this, we propose a feature weighting scheme combining genetic algorithm (GA) and support vector machines (SVMs) to estimate the weights for software metrics.

Due to containing rich information of programs, AST-based approaches are adapted efficiently to various problems with high accuracy [66, 92]. I surveyed different models from traditional ones to deep neural networks and found promising results even in the case of lazy learners. I also show an interest in applying this approach to source code categorization. Our proposals are to boost the quality of the learning. Firstly, AST data are refined by pruning redundant branches resulting in a significant reduction of AST nodes. This leads to not only an increase in the accuracy but also a decrease of the running time. To make more powerful models, we designed a sibling-subtree convolutional neural network to automatically extract ASTs' features. In addition, we present two types of combination models between deep neural networks and common learning algorithms.

An assembly code is a product after compiling a source file. Unlike ASTs that simply represent the structure, the assembly code reveals the behavior of the program since it contains atomic instructions executed sequentially. Besides, as mentioned before, software defect prediction is a challenging task because defect features are highly relevant to semantic meanings of programs. For these reasons, assembly code-based approaches may be beneficial to detect defects in software components. To validate this assumption, we propose and apply several deep neural networks to automatically learn defect features on assembly instruction sequences.

Our work in this thesis is illustrated in Fig. 1.4. Approaches for software engineering problems can be divided into three main directions based on: (1) software metrics-based that are suitable for the tasks of project document analysis such as software effort, cost, maintainability estimation; (2) AST-based that should be employed in analyzing program structures like source code classification and program plagiarism detection; (3) Assembly code-based that are appropriate to problems of discovering program actions including software defect prediction and malware analysis. To sum up, the main contributions of my research can be summarized as follows:

- **For software metrics-based approaches**. Proposing a hybrid model of GA and SVM for weighting the importance of software metrics to class labels.
- **For AST-based approaches**. Surveying several tree-based approaches for source code classification.
- Boosting the performance of tree-based approaches by (1) developing a sibling-subtree convolutional neural network on ASTs, (2) combining the neural networks and common machine learning algorithms, and (3) refining ASTs by pruning redundant branches.
- **For assembly code-based approaches**. Formulating end-to-end approaches that apply deep learning on assembly code of programs.
- Presenting an algorithm for constructing control flow graphs of assembly code.
- Designing a convolutional neural network for learning directed labeled graphs.

1.4 Dissertation Structure

The structure of this dissertation is as follows. Chapter 1 is an introductory chapter. It is started with describing several software engineering problems that can be solved by program analysis. Next, we discuss the traditional approaches and their limitations. I thereafter present the motivation and contributions of this research.

Chapter 2 provides the background of deep learning. Firstly, I describe the architecture of a simple network and the backpropagation algorithm for training the network. After that, I focus on clarifying convolutional neural network architectures which are mainly used to develop my proposed models.

Chapter 3 surveys traditional approaches for program analysis using machine learning and software metrics. To improve the performance, I propose a hybrid model of genetic algorithms and SVMs to simultaneously optimize the parameters and feature weighting/

selection. I conduct experiments on benchmark datasets for software defect prediction to verify the proposed model.

Chapter 4 presents approaches based on tree representations of source code written in programming languages with grammars. We apply machine learning techniques for tree structures including tree edit distance (TED), recursive neural networks, tree-based convolutional neural networks, and sibling-subtree convolutional neural networks. In addition, I proposed a combination of tree-based networks and kNN with TED to enhance the accuracies. The approaches are assessed on a task of source code classification by functionalities.

Chapter 5 aims to address tasks of source code analysis by applying graph-based approaches. Regarding this, each program is converted to a graph of control flow. We thereafter propose a graph-based convolutional neural network to build classifiers on graph datasets. The experiments are conducted on datasets of software defect prediction and malware analysis tasks.

Finally, we summarize the work in this dissertation mainly presented in Chapters 3, 4, and 5 including three directions of approaches for solving software engineering problems. For each direction, we have proposed several techniques to pre-process data and developed learning models to achieve remarkable performance.

Chapter 2

Deep Learning Background

2.1 Neural Networks

2.1.1 The Basic Definitions

A neural network is a computational model that is built from a set of interconnected processing elements, units or nodes called neurons. We will start with describing a basic element that is used to construct neural networks. Fig. 2.3 shows the components of a single neuron including inputs, an activation function, and an output. The inputs come from the other neurons or external sources, in which each one is associated with a weight indicating its relative importance to other inputs. To produce the output, the neuron applies a function f to the weighted sum of the inputs. Given an input x in $n-$dimensional space, the corresponding output is computed by the following function:

$$y = f(W^T \cdot x + b) \tag{2.1}$$

In the case of Fig. 2.3, the neuron takes two numerical inputs x_1 and x_2 with the weights w_1 and w_2. Additionally, there is another input **1** with the weight **b** called the bias. The role of the bias is to allow shifting the activation function to the left or right.

The activation function f is to transform non-linearly from the inputs into the output. There are a number of activation functions. They should have some desirable properties as follows:

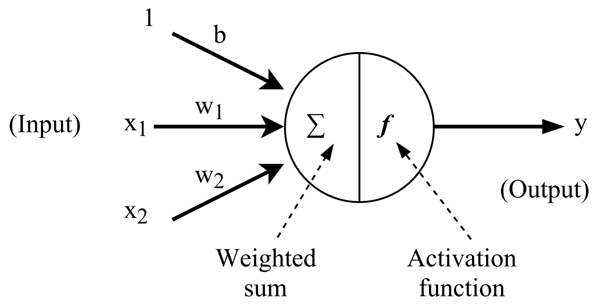


Figure 2.1: A single neuron

- Nonlinear. This is an important property that allows networks able to learn complex functions. Indeed, a two-layer neural network with nonlinear activation functions is proven to be a universal approximator. Meanwhile, a multi-layer network that all its neurons use linear activation functions is just equivalent to a single-layer model.
- Continuously differentiable. The most common algorithm for training neural networks is gradient descent. The gradient descent finds the minimum of a function by taking steps proportional to the inverse direction of the gradient at the current point, in which the gradient is a multi-variable generalization of the derivative. Thus, the continuously differentiable property is necessary to enable us to apply gradient-based optimization methods. Some activation functions may not differentiable at certain points. In these cases, gradient-based methods make no operation on the non-differentiable points.
- Monotonic. This is to guarantee that the error surface associated with a single-layer model is convex.

Fig. 2.2 plots some commonly used activation functions including *sigmoid*, *tanh*, and *ReLU*. Their equations are listed below:

- *sigmoid* maps any sized inputs to outputs in range [0,1]

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.2}$$

- *tanh* maps input to output ranging in [-1,1]

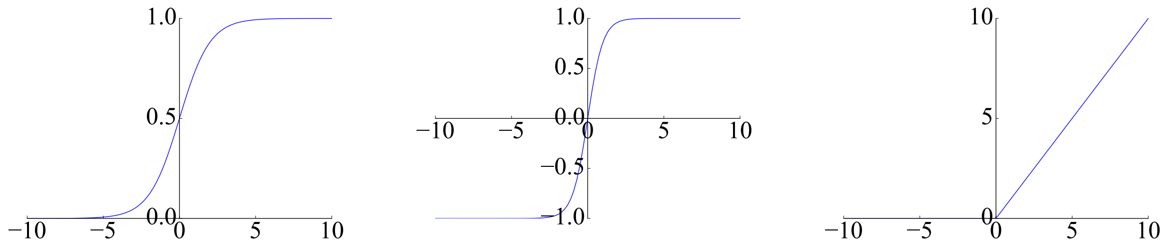
$$tanh(x) = \frac{2}{1+e^{-2x}} - 1 = 2\sigma(2x) - 1 \tag{2.3}$$

- *ReLU* removes negative part of function

$$ReLU(x) = max(0, x) \tag{2.4}$$

A neural network is a collection of such single neurons arranged in layers, in which the output of a neuron can be the input of another. Neurons between two adjacent layers have connections among them and each connection is associated with a weight. Fig. 2.3 shows a simplest type of neural networks called a feedforward neural network. The network receives input data from other sources at the first layer, performs different transformations, and produces the outputs at the last layer. The activation of a neuron i is the sum of the inner product of its inputs and connection weights, and the bias: $a_i = f(W_i \cdot x + b_i)$. Assume that the layers are fully-connected wherein every neuron in a layer is connected to all neurons in another layer. This means that the neurons in a layer have the same set of inputs, and we can write the activation computations for neurons in the layer in matrix notation as:

$$z = Wx + b \tag{2.5}$$



(a) Sigmoid (b) Tanh (c) ReLU

Figure 2.2: Some examples of activation functions

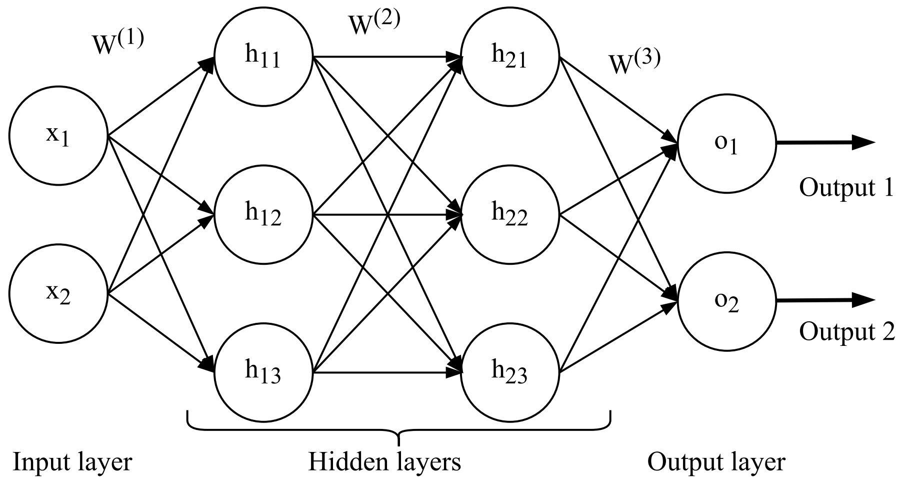


Figure 2.3: The architecture of a feedforward neural network with one hidden layer

$$a = f(z), \tag{2.6}$$

where $x \in \mathbb{R}^n$, $W \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$ and f is applied element-wise:

$$f(z) = f([z_1, z_2, ..., z_m]) = [f(z_1), f(z_2), ..., f(z_m)]. \tag{2.7}$$

In a feedforward network, the data is transformed and transferred in one direction from the input layer, through the hidden layers, and to the output layer. There are no cycles or loops in the network. The input nodes are responsible for receiving the information from outside and just passing them to the hidden nodes without any computation. The hidden nodes perform different types of transformations step by step through layers to the output. The computation process can be formulated as follows:

$$\begin{aligned}
z^{(2)} &= W^{(1)}x + b^{(1)} \\
a^{(2)} &= f(z^{(2)}) \\
z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\
a^{(3)} &= f(z^{(3)}) \\
z^{(o)} &= W^{(3)}a^{(3)} + b^{(o)} \\
y = a^{(o)} &= f(z^{(o)})
\end{aligned} \tag{2.8}$$

The output of the last layer is the result of the transformation done by the network model. Observing these values, we can evaluate the error of the model according to a certain measure. To obtain the optimal model, a set of data samples needs to be provided, and the model parameters are adjusted during a training procedure with the aim of minimizing the error made by the model on such dataset. In the next section, I will describe the general procedure for training neural networks, called backpropagation.

2.1.2 Training Neural Networks

The goal of training a neural network is to find a set of weights and biases such that for each input vector the network model can produce the desired output. To do that, we need to define an error function (sometimes known as the cost function or loss function) that maps each output vector to a real number. During the training process, the trainer makes efforts to minimize such error function to obtain the optimal model. Considering supervised learning, each sample is assigned a true label. The error of the network for each sample can be computed based on comparing the output and the true label. Various functions can be used the measure the error of a network model. Selecting a suitable function plays a critical role in training the network. A standard choice is the mean square error (MSE) that estimates the difference of the output and desired vectors by Euclidean distance. The MSE for a single sample and for n samples as computed by the Eq. 2.9 and Eq. 2.10.

$$MSE_i = \frac{1}{2}(\hat{y_i} - y_i)^2, \tag{2.9}$$

$$MSE = \frac{1}{2n}\sum_{i=1}^{n}(\hat{y_i} - y_i)^2 \tag{2.10}$$

where $\hat{y_i}$ is the vector predicted by the network, and y_i is the expected vector.

An efficient algorithm for optimizing neural networks is backpropagation using gradient descent method. The method performs an iteration procedure to adjust the network parameters such as the error function converges to the minimum value. At each step, we determine the gradient of the function at the current point by the following equation:

$$\nabla E = (\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, ..., \frac{\partial E}{\partial w_l}) \tag{2.11}$$

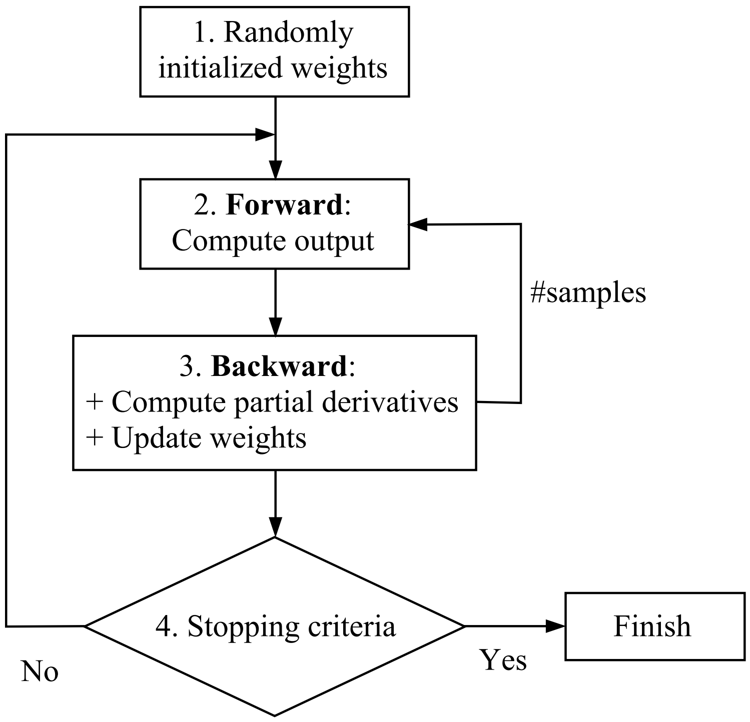

1. Randomly initialized weights
2. **Forward**: Compute output
#samples
3. **Backward**: + Compute partial derivatives + Update weights
4. Stopping criteria
No
Yes
Finish

Figure 2.4: Backpropagation algorithm

After that, each weight is adjusted by an amount proportional to the negative of the partial gradient:

$$\Delta w_i = -\gamma \frac{\partial E}{\partial w_i}, \text{ for } i = 1, ..., l \tag{2.12}$$

where l is the number of weights, and γ is the learning rate to control the convergence speed to the minimum value.

Since transforming from the input to the output is performed by a sequence of function compositions, the partial derivatives of the error function with respect the weights cannot be taken directly. In this case, the backpropagation algorithm is utilized to follow the inverse direction of the feedforward process to compute gradients for all the weights. The Fig. 2.4 indicates four main steps of the backpropagation algorithm. Firstly, the model parameters are randomly initialized. For each iteration, we traverse all the samples and do forward and backward steps for each sample. In forward step, the output of the sample is computed according to equation 2.8. The backward step computes the partial derivatives using the chain rules (Eq 2.11), and updates the weighs accordingly (Eq. 2.12). The training procedure finishes when satisfying the stopping criteria.

To illustrate the backward step, we will take an example of the feedforward network in figure 2.3 with its computations described in equation 2.8. Let consider a single weight of the connection between the i^{th} neuron in the penultimate layer and the j^{th} neuron of the output layer, its derivative can be easily computed by the chain rule:

$$
\begin{aligned}
\frac{\partial E}{\partial W_{ij}^{(3)}} &= \frac{\partial E}{\partial f(z_j^{(o)})} \frac{\partial f(z_j^{(o)})}{\partial W_{ij}^{(3)}} \\
&= \frac{\partial E}{\partial f(z_j^{(o)})} \frac{\partial f(z_j^{(o)})}{\partial z_j^{(o)}} \frac{\partial z_j^{(o)})}{\partial W_{ij}^{(3)}} \\
&= \frac{\partial E}{\partial f(z_j^{(o)})} \frac{\partial f(z_j^{(o)})}{\partial z_j^{(o)}} a_i^{(3)} \\
&= E'(f(z_j^{(o)}))f'(z_j^{(o)})a_i^{(3)} \\
&= \delta_j a_i^{(3)}
\end{aligned}
\tag{2.13}
$$

The equation 2.13 is computable because the functions are differentiable and the values are precomputed in the forward step. The full gradient of $W^{(3)}$ is written as follows:

$$
\frac{\partial E}{\partial W^{(3)}} = \delta^{(o)} a^{(3)^T} \tag{2.14}
$$

where $\delta^{(o)} \in \mathbb{R}^n$, with n is the number of neurons in the output layer. $\delta^{(o)}$ and $a^{(3)}$ can be seen as the error signals and input signals of the output layer. The derivatives for biases are computed similarly:

$$
\frac{\partial E}{\partial b^{(o)}} = \delta^{(o)} \tag{2.15}
$$

The chain rule can be applied in the same manner for the connection weights of low layers. Indeed, the partial derivatives of $W^{(2)}$ are written by the following equation:

$$
\begin{aligned}
\frac{\partial E}{\partial W^{(2)}} &= \frac{\partial E}{\partial f(z^{(3)})} \frac{\partial f(z^{(3)})}{\partial W^{(2)}} \\
&= \frac{\partial E}{\partial f(a^{(3)})} \frac{\partial f(a^{(3)})}{\partial f(z^{(3)})} \frac{\partial f(z^{(3)})}{\partial W^{(2)}} \\
&= \frac{\partial E}{\partial f(z^{(o)})} \frac{\partial f(z^{(o)})}{\partial f(a^{(3)})} \frac{\partial f(a^{(3)})}{\partial f(z^{(3)})} \frac{\partial f(z^{(3)})}{\partial W^{(2)}} \\
&= \frac{\partial E}{\partial f(a^{(o)})} \frac{\partial f(a^{(o)})}{\partial f(z^{(o)})} \frac{\partial f(z^{(o)})}{\partial f(a^{(3)})} \frac{\partial f(a^{(3)})}{\partial f(z^{(3)})} \frac{\partial f(z^{(3)})}{\partial W^{(2)}} \\
&= \delta^{(o)} W^{(3)} f'(z^{(3)}) a^{(2)^T} \\
&= \delta^{(3)} a^{(2)^T}
\end{aligned}
\tag{2.16}
$$

From equations 2.14 and 2.16, the error signals of the output layer can be re-used to compute the derivatives of the penultimate layer. One trick for network implementations is that storing the error signals at each layer to reduce the computations in the backward process.

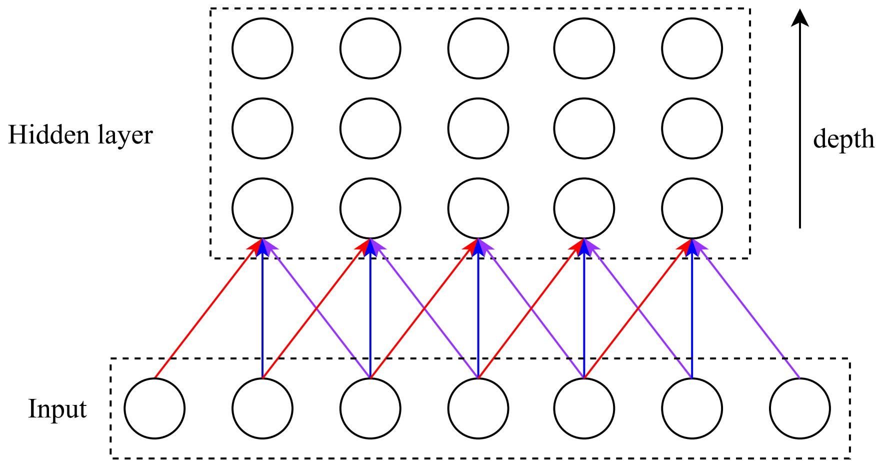


Figure 2.5: Illustration of local connectivity, two spatial dimensions and shared weights

2.2 Convolutional Neural Networks

2.2.1 Introduction

Convolutional neural networks (CNNs) are variants of multilayer feedforward neural networks using local connectivity, spatial arrangement, and shared weights. Fig. 2.5 illustrates an example of such variants. The connections with the same color have the same weight. The neurons in the hidden layer are arranged in two spatial dimensions. Each neuron is connected with a sub-region in the input layer spatially. The set of connection weights are replicated for every neuron in the same level. It should be noted that neurons along the depth connect to the same region by different weights. Hidden layers built by these manners are called convolutional layers that are the core blocks of a CNN.

Convolutional neural networks are extremely efficient for learning large-scale and high-dimensional inputs. In terms of the architecture, CNNs vastly reduce the number of parameters and avoid overfitting in comparison with regular networks. Taking the image processing task as an example, images have a size of $600 \times 600 \times 3$ respect with width, high, and color channels. For a regular network, each neuron in the first hidden layer has 600*600*3 = 1,080,000 weights, and the hidden layer mostly has many such neurons. As a result, the network contains a huge number of parameters that would lead to overfitting. Meanwhile, for a convolutional network with the local-region of size $20 \times 20 \times 3$, each neuron in the hidden layer only needs 20*20*3 = 1,200 weights. For learning performance, replicating weights across the dimensions allows CNNs be able to discover features regardless of their position in the input. In addition, we can construct a deep network by stacking multiple convolutional layers to learn high-level abstract features. Due to the tremendous advantages, convolutional neural networks have made many breakthroughs on numerous practical tasks such as programming language processing [66, 92], natural language processing [48], image processing [81], and so on.

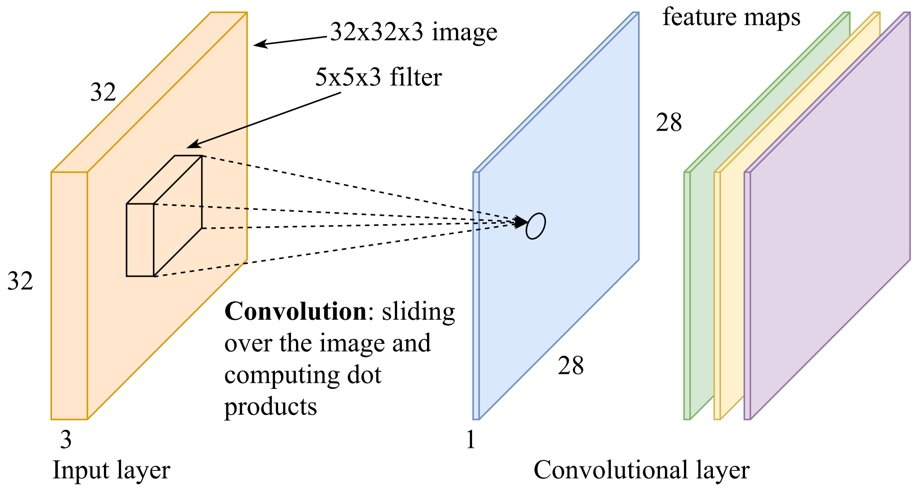


Figure 2.6: A convolutional layer with 5x5x3 filters for input images

2.2.2 The General Architecture

A simple CNN is a sequence of layers and it is designed specifically for handling high-dimensional inputs such as images, character sequences, and so on. The CNN architectures are formed by three main types of layers including convolutional layer, pooling layer, and fully-connected layer. In this section, we will describe in details the individual layers of CNNs on images. In next chapters, we will discuss several variations on other data structures like trees and graphs.

The convolutional layer is the core building block of a convolutional neural network. For image processing tasks, each image is represented as a three-dimensional array with respect to width, height, and three color channels (Red, Green, Blue). A sub-region is spatially along the width and height, and extends through the all channels. Fig. 2.6 shows a convolutional layer that takes an input image of size $32 \times 32 \times 3$ and connects to sub-regions of size $5 \times 5 \times 3$. The neurons are arranged in three dimensions. With respect to the above description, neurons along the width and height share a set of weights called a filter, and neurons along the depth look at the same region. The forward pass of this layer can be viewed as sliding each filter over the width and height of the image, and computing dot products between the filter and the input at any position. This process produces a two-dimensional activation map that corresponds to the set of neurons on a surface. The activation maps of the filters are stacked along the depth dimension to form the convolutional layer. We can see that the convolutional layer transforms from the input to the output with the same dimensions (width, height, depth). Thus, we can stack another convolutional layer on the top of the output in the same manner to perform more abstract transformations.

The number of neurons and spatial arrangement of a convolutional layer are controlled by three parameters including the depth, stride, and zero-padding. Next, we will discuss these parameters.

- The depth corresponds to the number of filters we intend to use. Each filter works

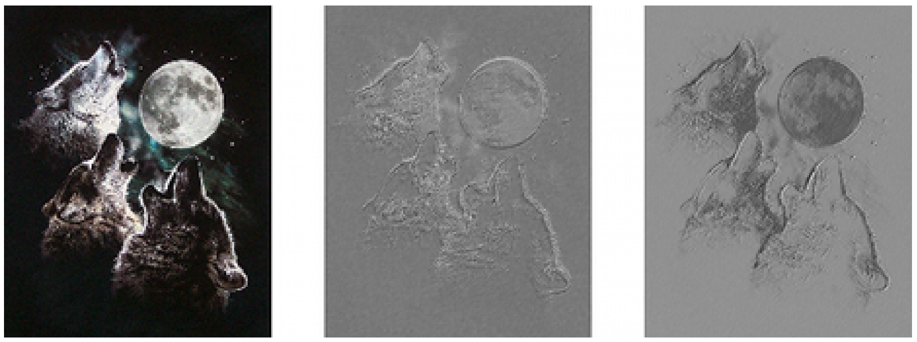


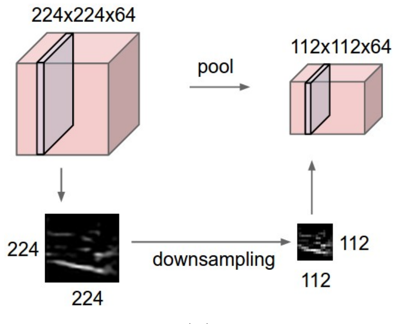
Figure 2.7: The feature maps of two filters on an input image

as a feature extractor to detect certain types of visual features on the input image such as an edge of an orientation, circle-like patterns. Fig. 2.7 shows an example of applying two filters on an input image to produce feature maps. Depending on each problem, we should determine the suitable number of filters such that extracted features are enough to capture the objects on images.

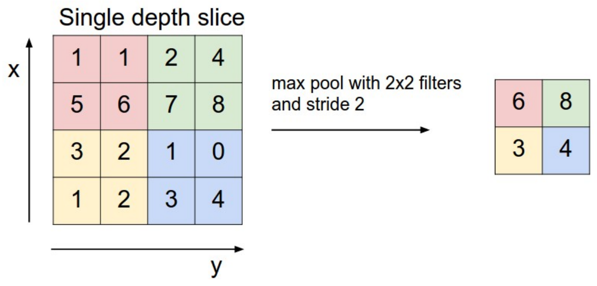
- The stride is the number values that the filter slides over the input spatially. The stride being S means that the filters jump S pixels at a time when we slide horizontally or vertically over the image. The greater stride produces the smaller feature map size. In practice, smaller strides work better.
- The zero-padding is adding zeros around the borders. As can be seen in Fig. 2.6, the size of a feature map is smaller than that of the input. Without padding, the information at the borders is lost quickly after several convolution layers. Therefore, the zero-padding should be used to control the spatial size of the output. Commonly, we will use it to preserve the structures of the input such that the input and the output have exactly the same width and height.

Given the three parameters, the spatial size of the output can be computed by $(W - F + 2P)/S + 1$, where W, F, P and S are the input size, filter size, zero-padding and stride, respectively. Taking 2.6 as an example, the 32×32 input, 5×5 filters with stride 1 and pad 0, the size of each feature map is $(32 - 5 + 2 * 0)/1 \times (32 - 5 + 2 * 0)/1 = 28 \times 28$. This means that each surface includes $28 \times 28 = 784$ neurons arranged in two dimensions 28×28.

Pooling layer In a CNN, a pooling layer is commonly inserted between two convolutional layers to perform dimension reduction. Convolutional layers are responsible to extract features using a set of filters, but they preserve the structure of the input. After a convolution, the spatial size of the output is similar to that of the input, or exactly the same in the case of using zero-padding. For high-dimensional data, dimension reduction is necessary to reduce the parameters and computation in the network to control overfitting. This is implemented in convolutional neuron networks by stacking a pooling layer of the top of each convolutional layer. The pooling layer resizes spatially, independently



(a)



(b)

Figure 2.8: A max pooling layer to reduce the dimensions spatially, independently in each depth slice of the input.

each feature map by using an operation such as *max*, *average*, or *L2-norm*. The depth dimension is unchanged.

Generally, for image processing tasks, a pooling layer takes inputs of size $W_1 \times H_1 \times D_1$, and produces the outputs of size $W_2 \times H_2 \times D_2$, in which the output size is determined as follows:

- $W_2 = (W_1 - F)/S + 1$
- $H_2 = (H_1 - F)/S + 1$
- $D_2 = D_1$

where F and S are the filter size and stride of the pooling layers. Fig. 2.8 shows a max pooling layer with the filter size of 2×2 and the stride of 2. This layer reduces a half of the spatial size (width and height) and preserves the depth dimension of the input. For each depth slice, the max operation is applied to all non-overlapping regions of size 2×2 and picks the max of the four values in each region. It is worth noting that zero-padding is not commonly used in pooling layer, and the little regions are overlapping in the case of $F > S$ and non-overlapping in the case of $F = S$.

Back-propagation for a convolutional layer or a pooling layer is applied in the similar way to that of a feedforward layer. During the backpropagation of the convolutional layer, every neuron computes the gradient for its weights. These gradients are added up across each depth slice, and updating a single set of weights is performed per slice.

For a pooling layer, the forward pass reduces each $N \times N$ block to a single value. The backpropagation routes gradient to the inputs that produce the output in the forward pass. In the case of max-pooling, the error is just passed to the previous layer at the neuron with the max value. Since the other neurons in the pooling block did not contribute to the output, their errors are assigned 0. For average pooling, the error is multiplied by $\frac{1}{N \times N}$ and assigned to all the neurons in the pooling block.

2.3 Evaluation Measures

To compare the performance of machine learning approaches, we will use a set of evaluation measures. Most of the common measures can be computed from the confusion matrix that is derived from observing the outputs. The confusion matrix is a table, in which each row represents the number of instances in a predicted class and each column represents the number of instances in an actual class (or vice versa). Table 2.1 shows the confusion matrix for binary classification problems that data samples are classified into two categories of positive and negative labels. There are four possible outcomes when predicting an instance using a classifier. If both the actual and predicted labels are positive, the instance is counted as a true positive; if the actual label is positive and the predicted label is negative, the instance is counted as a false negative; if the actual label is negative and the predicted label is positive, the instance is counted as a false positive; if both the actual and predicted labels are negative, the instance is counted as a true negative. Based on the confusion matrix, several evaluation measures are computed as follows:

Table 2.1: The confusion matrix

		Predicted	
		1	-1
Actual	1	True Positive (TP)	False Negative (FN)
	-1	False Positive (FP)	True Negative (TN)

- **True positive rate (TPR)**, also called recall, hit rate of sensitivity, is the ratio of correctly predicted positive instances to total positive instances.

$$TPR = recall = \frac{TP}{TP + FN} \tag{2.17}$$

- **True negative rate**, also called specificity, is the ratio of correctly classified negative instances to total negative instances.

$$TNR = \frac{TN}{TN + FP} \tag{2.18}$$

- **False positive rate**, also called false alarm rate, is the ratio of incorrectly classified negatives to total negative instances.

$$FPR = \frac{FP}{FP + TN} \tag{2.19}$$

- **Precision** is the ratio of correctly predicted positive instances to the total predicted positive instances.

$$Precision = \frac{TP}{TP + FP} \tag{2.20}$$

- **Accuracy** is the ratio of correctly classified instances to the total instances.

$$Accuracy = \frac{TP + TN}{TP + FN + FP + TN} \tag{2.21}$$

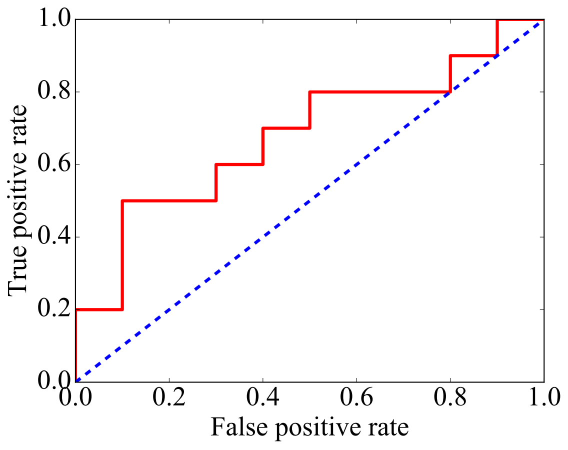
- **F1 score** is the harmonic average of precision and recall. This measure takes into account both precision and recall to compute the score.

$$F1 = 2\frac{Recall * Precision}{Recall + Precision} \tag{2.22}$$

In machine learning, the area under the receiver operating characteristic (ROC) curve, known as the AUC, that 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 [25]. According to previous research, AUC has been proved as a better and more statistically consistent criterion than the accuracy [55], especially for imbalanced data. In the cases of imbalanced datasets 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 [75].

#Inst	Class	Score	#Inst	Class	Score
1	p	.9	11	p	.4
2	p	.8	12	n	.39
3	n	.7	13	p	.38
4	p	.6	14	n	.37
5	p	.55	15	n	.36
6	p	.54	16	n	.35
7	n	.53	17	p	.34
8	n	.52	18	n	.33
9	p	.51	19	p	.30
10	n	.505	20	n	.1

(a)



(b)

Figure 2.9: An example of ROC curve plotted according to the outputs of a classifier.

The AUC is estimated by the area under the ROC curve that depicts the tradeoffs between hit rates and false alarm rates. Fig. 2.9 illustrates plotting the ROC curve for a probabilistic classifier that outputs scores to assess the degrees of the instance belonging to classes. With a threshold value, if the classifier output is above the threshold, the predicted label is positive; otherwise, is negative. We thereafter compute the TPR and FPR based on the confusion matrix to create a point in ROC space. By varying the threshold from the min to the max values of scores and connecting all the points, we will obtain the ROC curve.

To extend the use of ROC curves to multi-class problems, the average results are computed based on two ways: 1) macro-averaging gives equal weight to each class, and 2) micro-averaging gives equal weight to the decision of each sample [85]. According to the AUC measure, the higher value the better discrimination ability the classifier has.

Chapter 3

Software Metrics-based Approaches

This chapter presents a traditional approach for program analysis based on software metrics. While current methods consider software metrics with the same role in learning models, my research discovers the importance extent for each metric according to its relevance the class labels. To do this, I proposed a combination model of genetic algorithms (GAs) and support vector machines (SVMs) to simultaneously optimize the classifier parameters and metrics weights.

3.1 Software metrics

A software metric is a quantitative measure of a given attribute of a system, component, or process. Table 3.2 describes 17 metrics of `File 1.c` written in `C` language. The metric values are obtained by using a standard tool for source code metrics and quality analysis, namely RSM [1]. Analyzing software metrics brings many advantages to software development. For example, with a defect prediction system, developers will pay more attention to potentially defective modules to localize and fix bugs. This enhances software quality and reduces the development time and the product cost as well. Estimating the level of complexity supports managers in schedule and budget planning, cost estimation, and optimal personnel task assignments.

```
File 1.c
#include<stdio.h>
int gcd(long long int a,long long int b)
{
   if(b!=0)
      return gcd(b,a%b);
   else
      return a;
}
int main()
{
```

[1]http://msquaredtechnologies.com/m2rsm/

```
    int t;
    scanf("%d",&t);
    while(t--)
    {
        long long int q,p;
        scanf("%lld %lld",&q,&p);
        long long int h=gcd(q,p);
        long long int l=(p*q)/h;
        printf("%lld %lld\n",h,l);
    }
}
```

Table 3.1: List of software metrics of source code files

No.	Metrics	Meaning	Value
1	File Function Count	Number of functions	2
2	Total Function LOC	Lines of Code	18
3	Total Function Pts LOC	Function Points Derived from LOC metrics	0.4
4	Total Function eLOC	Effective LOC	12
5	Total Function Pts eLOC	Function Points Derived from lLOC metrics	0.3
6	Total Function lLOC	Logical Statements LOC	9
7	Total Function Pts lLOC	Function Points Derived from lLOC metrics	0.2
8	Total Function Params	Number of Input Parameters	2
9	Total Function Return	Number of Return Points	3
10	Total Cyclo Complexity	Cyclomatic Complexity Logical Branching	4
11	Total Function Complex	Functional Complexity (Interface + Cyclomatic)	9
12	Max Function LOC	Max LOC of functions	12
13	Average Function LOC	Average LOC of functions	9.0
14	Max Function eLOC	Max eLOC of functions	8
15	Average Function eLOC	Average eLOC of functions	6.0
16	Max Function lLOC	Max lLOC of functions	7
17	Average Function lLOC	Average lLOC of functions	4.5

Many software metrics have been proposed over a period of time. They can be classified into three categories: product metrics, process metrics, and project metrics. Product metrics describe the characteristics of the product such as size, complexity, design features, performance, and quality level. Process metrics are a collection of software-related activities including the effectiveness of defect removal during development, the pattern of testing defect arrival, and the response time of the fixing process. Project metrics describe the project characteristics and execution, for instance, the number of software developers, the staffing pattern over the life cycle of the software, cost, schedule, and productivity. Some metrics belong to multiple categories. For example, the in-process quality metrics of a project are both process metrics and project metrics [43]

3.2 Feature weighting using genetic algorithms

3.2.1 Introduction of Support Vector Machines (SVMs)

This section will briefly describe an effective classification algorithm in supervised machine learning called Support Vector Machines (SVMs) [64]. Firstly, we introduce the algorithm for separable datasets, then present its general version designed for non-separable datasets, and finally provide a theoretical foundation for SVMs based on the notion of margin. We start with the description of binary (two-class) classification problems in the separable case.

The optimal hyperplane for separable data

Given a training set $S = \{x_i, y_i\}_{i=1}^{m}$, with input vectors $x_i \in R^n$ and target labels $y_i \in \{-1, +1\}$, for the linearly separable case, the data points will be correctly classified by any hyperplanes $w \cdot x + b = 0$ satisfying

$$y_i(x_i \cdot w + b) - 1 \geq 0,\ \forall i = 1, .., m \tag{3.1}$$

Although many hyperplanes perfectly separate the training samples into two classes, SVMs find an optimal separating hyperplane with the maximum margin (distance to closest points) by solving the following optimization problem:

$$\min_{w,b} \frac{1}{2}||w||^2 \tag{3.2}$$

subject to $y_i(w \cdot x_i + b) \geq 1, \forall i \in [1, m]$. This quadratic optimization problem can be solved by finding the saddle point of the Lagrange function:

$$L\,(w,\ b,\ \alpha) = \ \frac{1}{2}||w||^2 - \sum_{i=1}^{m} \alpha_i [y_i\,(w \cdot x_i + b) - 1] \tag{3.3}$$

where α_i denotes Lagrange variables, $\alpha_i \geq 0\ \forall\ i = 1, ..m$.

The Karush Kuhn -Tucker (KKT) conditions for a maximum of Eq.(3.3) are obtained by setting the gradient of the Lagrangian with respect to the primal variables w and b to zero and by writing the complementary conditions:

$$\nabla_w L = w - \sum_{i=1}^{m} \alpha_i y_i x_i \ = 0 \ \Rightarrow\ w = \sum_{i=1}^{m} \alpha_i y_i x_i \tag{3.4}$$

$$\nabla_b L = -\sum_{i=1}^{m} \alpha_i y_i \ = 0 \ \Rightarrow \sum_{i=1}^{m} \alpha_i y_i \ = 0 \tag{3.5}$$

$$\forall i,\ \alpha_i\,[y_i\,(w \cdot x_i + b) - 1] = 0 \ \Rightarrow \alpha_i = 0\ \vee\ y_i\,(w \cdot x_i + b) - 1 = 0 \tag{3.6}$$

By Eq.(3.4), the weight vector w solution of the SVM problem is a linear combination of the training set vectors $x_1, ..., x_m$. According to complementary conditions (3.6), the value of w only depends on vectors x_i that correspond the $\alpha_i \neq 0$. Such vectors are called support vectors. They fully define the maximum-margin hyperplane or the SVM solution.

Substitute Eqs. (3.4) and (3.5) into Eq. (3.3), the dual form Lagrangian $L_D(\alpha)$ of Eq. 3.2 is derived as follows:

$$\max_{\alpha} L_D(\alpha) \ = \sum_{i=1}^{m} \alpha_i \ - \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j (x_i \cdot x_j) \tag{3.7}$$

subject to $\alpha_i \geq 0 \ i = 1, .., m$ and $\sum_{i=1}^{m} \alpha_i y_i = 0$.

To find the optimal hyperplane, $L_D(\alpha)$ must be maximized with respect to non-negative α_i. The objective function is a standard quadratic optimization problem that can be solved by using several standard optimization methods. The solution α can be used directly to determine the parameters w^* and b^* of the optimal hyperplane returned by the SVM. Thus, we obtain an optimal decision hyperplane $f(x)$ (Eq.(3.8)) and an indicator decision function $sign[f(x)]$.

$$f(x) = \sum_{i=1}^{m} \alpha_i^* y_i \ (x_i \cdot x) + b^* = \sum_{i \in SV} \alpha_i^* y_i \ (x_i \cdot x) + b^* \tag{3.8}$$

where b^* is calculated based on rewriting condition Eq.(3.6) as follows:

$$b^* = y_i - \sum_{j=1}^{m} \alpha_j y_j (x_j \cdot x_i) \tag{3.9}$$

The optimal hyperplane for non-separable data

In most practical settings, data are often not linearly separable [64]. For any hyperplane $w \cdot x + b = 0$, there exists x_i such that

$$y_i(x_i \cdot w + b) - 1 \ngeq 0 \tag{3.10}$$

In this case, an SVM selects a hyperplane that minimizes the training error. The constraints in Section 3.2.1 cannot all hold simultaneously, but the above concepts can be extended to the non-separable case. To get the formal setting of this problem, non-negative slack variables ξ_i are proposed such that:

$$y_i(x_i \cdot w + b) \geq 1 - \xi_i, \ \xi_i \geq 0, \ i = 1, .., m \tag{3.11}$$

Here, a slack variable ξ_i measures the distance by which vector x_i violates the desired inequality, $y_i(w \cdot x_i + b) \geq 1$. For a hyperplane $w.x + b = 0$, a vector x_i with $\xi_i > 0$ can be viewed as an outlier. An SVM finds the optimal hyperplane by minimizing the expression below:

$$\min_{w,b,} \frac{1}{2} ||w||^2 \ \ + \mathrm{C} \sum_{i=1}^{m} \xi_i \tag{3.12}$$

subject to $y_i(w \cdot x_i + b) \geq 1 - \xi_i \wedge \xi_i \geq 0, \ i \in [1, m]$

Minimizing the expression (3.12) is an NP-hard problem. There are two conflicting objectives: seeking a hyperplane with larger margin and limiting the total amount of slack variables measured by $\sum_{i=1}^{m} \xi_i$. The parameter $C \geq 0$ is known as trade-off between two such objectives. Typically, C is determined via k-fold cross validation method.

The optimization model can be solved by maximizing the dual variables Lagrangian $L_D(\alpha)$ (Eq.(3.13)), which only differs from that of the separable case Eq.(3.7) by the constraints $\alpha_{\ i} \leq C$:

$$\max_{\alpha} L_D(\alpha) \ = \sum_{i=1}^{m} \alpha_i \ - \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j (x_i \cdot x_j) \tag{3.13}$$

subject to $0 \ \leq \alpha_i < C, \ i = 1, .., m \wedge \sum_{i=1}^{m} \alpha_i y_i = 0$.

Two parameters w and b of the optimal hyperplane can be determined directly via solution α similar to separable case (Eqs.(3.4), (3.9)). However, support vectors in non-separable case include outliers and vectors which lie on marginal hyperplanes.

Non-linear SVM

The main idea of creating non-linear kernel classifiers is mapping the data into a higher-dimensional feature space in the hope that in the higher-dimensional space the data could become more easily separated or better structured. This is performed by using a mapping function Φ and replacing the dot products in Eq.(3.13) by the kernel function (3.14):

$$K(x_i, x_j) = (\Phi(x_i), \Phi(x_j)) \tag{3.14}$$

$$\max_{\alpha} L_D(\alpha) \ = \sum_{i=1}^{m} \alpha_i \ - \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j K(x_i, x_j) \tag{3.15}$$

subject to $0 \ \leq \alpha_i < C, \ i = 1, .., m \wedge \sum_{i=1}^{m} \alpha_i y_i = 0$

Some widely used kernel functions include polynomial, radial basis function (RBF) and sigmoid kernel, which are shown as functions (3.16), (3.17) and (3.18). Choosing the most appropriate kernel function and its parameters are completely based on the specific dataset. There are various methods to determine parameters in kernel functions. In this work, we use the genetic algorithm to find the optimal values of parameters and weights of data attributes.

- Polynomial kernel:
$$K(x_i, x_j) = (1 + x_i \cdot x_j)^d \tag{3.16}$$
- Radial basis function kernel (alternative form):
$$K(x_i, x_j) = exp(-\gamma ||x_i - x_j||^2) \tag{3.17}$$
- Sigmoid kernel:
$$K(x_i, x_j) = tanh(k x_i \cdot x_j - \delta) \tag{3.18}$$

3.2.2 Genetic algorithms - GAs

Genetic algorithms have been used in science and engineering as adaptive algorithms for solving practical problems [63]. The exploitation of the principles of evolution as a heuristic method enables genetic algorithms to solve optimization problems effectively (with the acceptable solutions) without using the traditional conditions (continuity or differentiability of the objective function) as prerequisites.

One of the most important characteristics of GAs is the ability to work with a population of individuals, each representing a feasible solution to a given problem. The search is now performed in parallel on multi-points. However, this is not a simply multi-points searching algorithm because the points are interactive with each others based on principles of natural evolution [29]. The basic steps of GAs (Fig. 3.1) are described as follows [71]:

- Step 1 : $t = 0$; Initialize $P(t) = \{x_1, x_2, ..., x_n\}$, where n is the number of individuals .
- Step 2: Calculate the value of the objective functions for $P(t)$.
- Step 3: Create a crossover pool $MP = se\{P(t)\}$ where se is selection operator.
- Step 4: Determine $P'(t) = cr\{MP\}$, with cr is the crossover operator.
- Step 5: Determine $P"(t) = mu\{P'(t)\}$, with mu is the mutation operator.
- Step 6: Calculate the value of the objective functions for $P"(t)$
- Step 7: Determine $P(t + 1) = P"(t)$ and set $t = t + 1$
- Step 8: Return Step 3, if the stop condition is not satisfied.

Solutions representation. This task plays a crucial role in designing genetic algorithms, deciding whether to apply the evolutionary operators. One of the traditional representations of GAs is the binary representation. In this way, a feasible solution to a problem is represented as a vector of bits called a chromosome. Each chromosome consists of many genes; a gene represents a parametric component of the solution. A different type of chromosome representation is using real numbers. With this representation, the evolution operators will perform directly on the real values(genes).

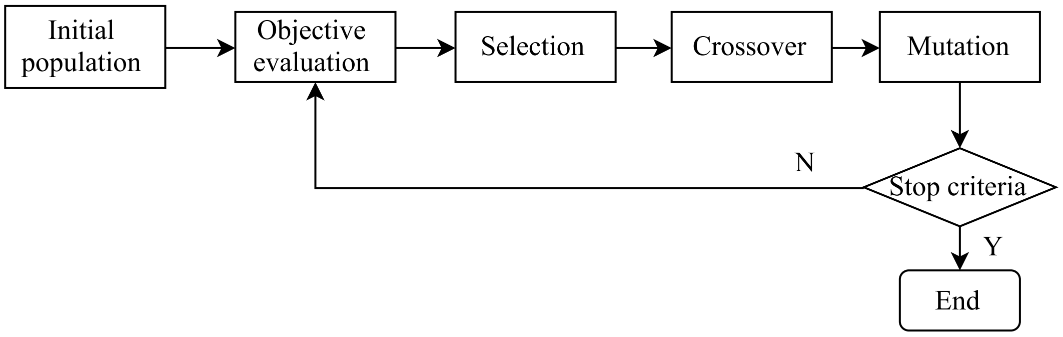


Figure 3.1: Basic steps of genetic algorithm

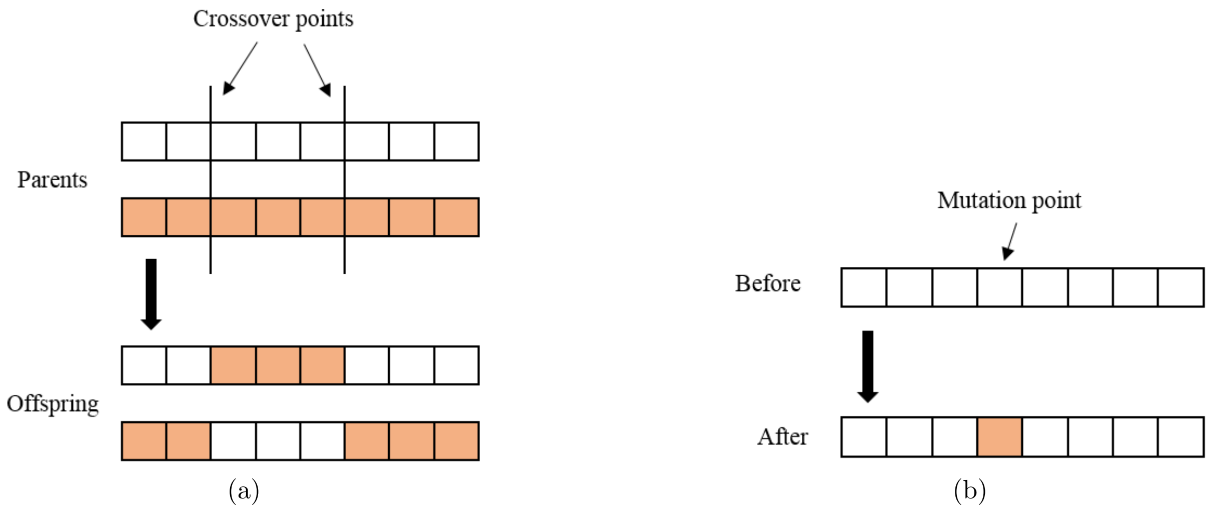


Figure 3.2: Illustration of the two-point crossover and mutation operators and their effects in the generation of the offspring

Selection. The goal of the selection stage is guiding the search towards better individuals and maintaining a high genotypic diversity in the population. The quality of each individual is evaluated by mean of the fitness function. This value is used to determine which individual will be selected for the next generation whereby the more greater quality the individual has the more chance it is chosen. Some commonly used selection methods include:

- Roulette wheel: Selecting individuals is based on probability (proportional to the value of the fitness function). Each is assigned a slice of a circular “roulette wheel", the size of the slice being proportional to the individual’s fitness. The wheel is spun N times, where N is the number of individuals in the population. On each spin, the individual under the wheel’s marker is selected to be in the pool of parents for the next generation [63].

- Tournament selection: The selection process involves running several “tournaments"

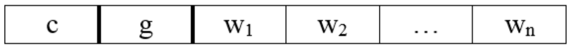


Figure 3.3: The structure of a chromosome for optimizing C, γ and weights of features

between two individuals chosen randomly from the population. The better individual (the winner) having the greater fitness value is selected for the next generation.

- Elitist selection: For this selection strategy, a limited number of individuals with the best fitness values are chosen for the next generation. Elitism avoids losing individuals with good genetics by crossover or mutation operators.

Crossover. Crossover operators are applied to generate new individuals from their parents. Crossover operators are inspired by the idea that offspring inherit the best characteristics from their parents. In terms of searching, crossover operators perform a search of the area around the solution represented by the parent individuals. There are some crossover techniques including single-point, two-point and uniform crossovers.

Mutation. Similar to crossover operators, mutation operators are also used to simulate mutation phenomena in biology. Mutations often generate new individuals different from their parents. In terms of searching, mutation operators aim to deploy the finding out of the local area. Fig. 3.2 illustrates the genetic crossover and mutation operators.

The evolutionary process is repeated until the stop criteria such as the pre-defined number of generations and an acceptable fitness value are satisfied [18,29] (Fig. 3.1).

3.2.3 Hybrid model GA-SVM for feature weighting and parameter optimization

In this section, we present in detail how to combine GA and SVM for feature weighting and parameter optimization involving chromosome design, fitness function, and system architecture. Our implementation was carried out on C# language by extending LIBSVM, which is originally designed by Chang and Lin (2001)[16].

Chromosome design

The proposed model searches kernel function parameters and the weights of features simultaneously. Hence, the chromosome must contain two such parts. In our experiments, we used the RBF kernel function for two reasons. Firstly, this kernel maps samples into a higher-dimensional space; hence, it can handle the case when the relation between class labels and attributes is nonlinear. The second reason is the number of hyperparameters, which influences the complexity of model selection. The RBF kernel only requires two parameters C and γ. Fig. 3.3 shows the structure of a chromosome in this case.

Real coding was used to represent the chromosome. All genes in the chromosome have value in the range [0, 1]. Two first genes c and g represent the values of C and γ respectively, while $w_1 \sim w_n$ represent the weights of features. Note that, we search values of C

and γ in the ranges $[C1,\ C2]$ and $[\gamma_1,\ \gamma_2]$. Thus, parameters C and γ of the SVM classifier are obtained by mapping c and g into corresponding ranges via the following formula:

$$C = C_1 + c * (C_2 - C_1)\ and\ \gamma = \gamma_1 + g * (\gamma_2 - \gamma_1) \tag{3.19}$$

In the implementation, we allow users to vary lower-bound and upper-bound values of C and γ as well as other parameters of the genetic algorithm.

Fitness function

The performance of SVM classifiers is used to design a single objective function for GAs. In the decoding process, both training and testing datasets are transformed by multiplying feature i^{th} with the corresponding weight $w_i,\ i = 1, .., n$. After that, the SVM model with the RBF kernel function is built based on $C,\ \gamma$ (Eq.(3.19)) and the transformed training dataset. The accuracy of the classifier on the testing dataset is used to assesses the quality of the chromosome.

In GAs, the objective functions are very important and they notably affect the rate of convergence and the quality of the best solutions [24]. In the model, we also provide different objective functions such as the accuracy, F1 score, or MCC(3.20).

$$MCC = 2 \cdot \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{3.20}$$

System architecture

We suggest an architecture for the hybrid system GA-SVM as Fig. 3.4. In this model, the task of GA is to search optimal parameters of SVM and weight of features. Besides, the SVM plays a role as oriented search strategy for GA by evaluating the fitness of chromosomes. The details of the hybrid model are described as follows:

1. *Data pre-processing (scaling).* Scaling before applying SVM is very important. The main advantage of scaling is to prevent attributes in greater numeric ranges dominating those in smaller numeric ranges. Another advantage is to avoid numerical difficulties during the calculation process [37]. When using the Grid algorithm, according to our experimental results, feature value scaling can help to increase SVM accuracy. Normally, each feature is linearly scaled to the range $[-1, +1]$ or $[0, 1]$ by formula (3.21).

$$v' = \frac{v - min_a}{max_a - min_a} \tag{3.21}$$

   where v' is scaled value, v is original value, min_a, max_a are low bound and upper bound of the feature value, respectively.

   Typically, we have to use the same method to scale both training and testing data. For example, suppose that we scaled the first attribute of training data from $[-10, +10]$ to $[-1, +1]$. If the first attribute of testing data lies in the range $[-11, +8]$, we must scale the testing data to $[-1.1, +0.8]$.

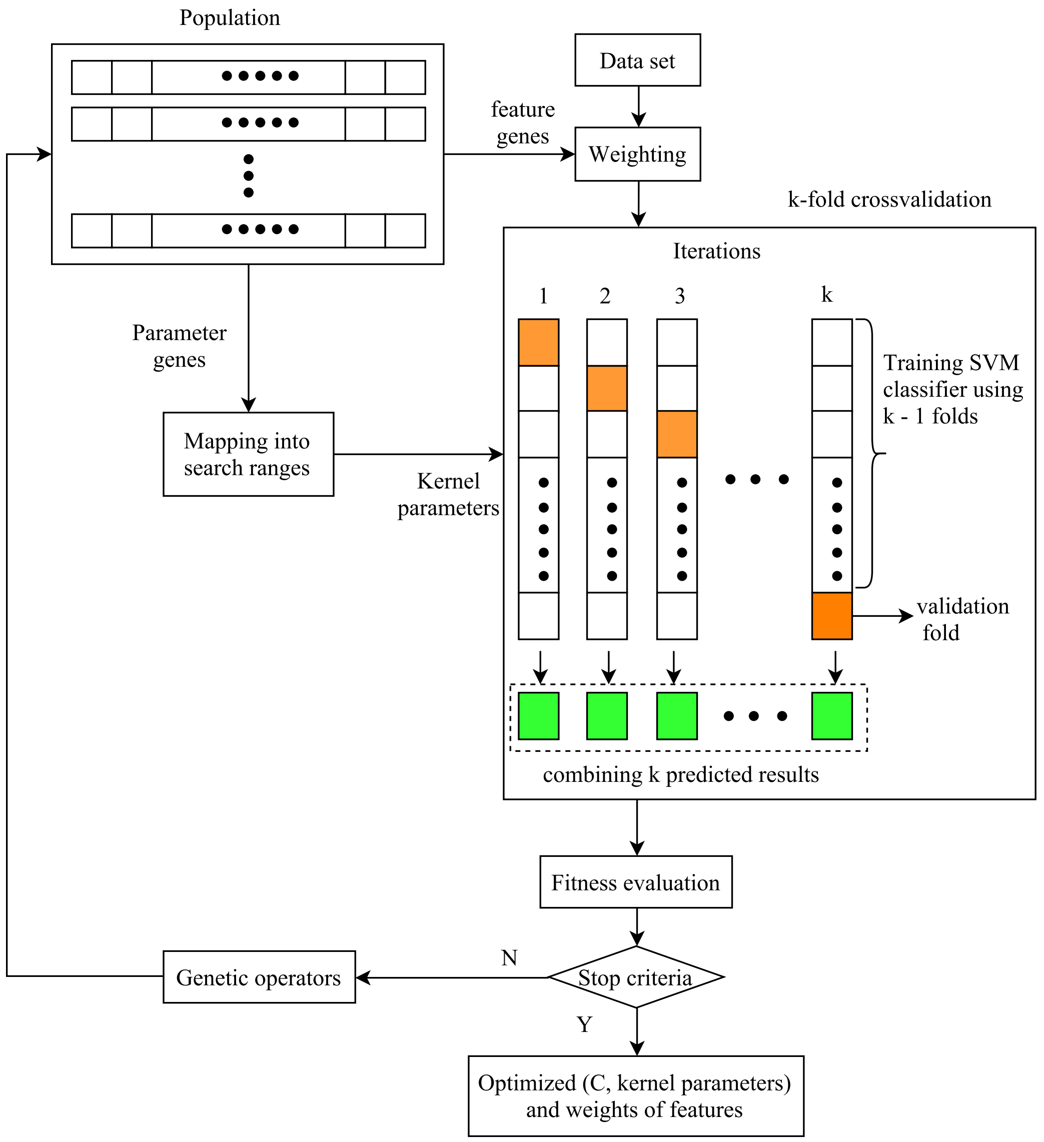


Figure 3.4: The system architecture of the hybrid GA-SVM model for feature weighting and parameter optimization

For the task of finding the weights of features, scaling data may be unnecessary for GA-SVM model. However, in order to gain the best results for the Grid algorithm, we performed this process. Then all experiments of both the approaches were conducted on the scaled datasets.

2. *Decoding(generating training data and parameters C, γ for SVM).* This step converts the values of the chromosome into parameters C, γ of the SVM by Eq(3.19) and the weights of features. In both the training and testing datasets, the feature values of instances are multiplied by the corresponding weights using following formula.
$$\bar{x}_{ij} = x_{ij} * w_j \tag{3.22}$$
where x_{ij} and $\bar{x}_{ij}$ are the value of the j^{th} field of the i^{th} instance before and after scaling. w_j is the weight of the j^{th} field.

3. *Fitness evaluation* After decoding stage, C, γ and the scaled training dataset are used to build the SVM model. The scaled testing dataset is used to evaluate the performance of the classifier. When the predicted data is obtained, each chromosome is evaluated by the fitness functions described in Section 3.2.3.

4. *Genetic operators.* In this step, a new generation is produced by genetic operators including selection, crossover, mutation, and replacement.

5. *Stopping criteria.* The population is improved through many generations. The evolutionary process ends when stopping criteria are satisfied. Some typical criteria are used such as a number of iterations, acceptable results or a fixed number of last generations without changing the fitness value.

6. *Elitism replacement.* To maintain the good solutions of each generation that may be lost during the evolutionary process by crossover and mutation operators, we use the elitism replacement technique. For each generation, we store the best chromosome and replace worst chromosome in the next generation with such chromosome.

3.3 Experiments

3.3.1 Datasets

The datasets for conducting experiments were collected from the NASA[2] (CM1, KC1, KC2, and KC3) and the PROMISE[3] (jEdit1 and jEdit2) repositories. They are extracted from practical software projects and publicly available in order to motivate verifiable and improvable predictive models of software engineering. Each data sample corresponds to a module in the project, and is labeled as a clean code or defective code.

[2]`http://openscience.us/repo/`

[3]`http://promisedata.org`

Table 3.2: Software metrics datasets

Dataset	Language	#Metrics	#Clean	#Defective	System
CM1	C	21	394	48	NASAspacecraft instrument
jEdit1	Java	8	136	134	jEdit version 4.0_4.2
jEdit2	Java	8	161	202	jEdit version 4.2_4.3
KC1	C++	21	897	315	Storage management
KC2	C++	21	270	105	Scientific data processing
KC3	C++	39	283	43	NASA MDP version

Table 3.2 shows statistics on the six datasets. jEdit1 and jEdit2 are calculated from versions 4.0, 4.2 and 4.3 of the jEdit system, a well-known text editor written in Java. They contain six attributes for object-oriented projects, number of public methods and number of lines of code. The attributes of the other datasets include Halstead Complexity, McCabe's complexity, and those computed from statistical values of source code such as operators, operands, number of lines of code, etc. As can be seen, most of the datasets are imbalanced since the numbers of clean modules are much greater than that of defective modules. This is a big challenge for any machine learning algorithm.

3.3.2 Experimental setup

The detailed settings for the genetic algorithm are as follows: population size 500, crossover rate 0.9, mutation rate 0.05, two-point crossover, elite selection and elitism replacement. In addition, we set the ranges of C [0.01 - 32000] and γ [10^{-6} - 8]. The aim of such ranges is to limit the searching bounds of two SVM parameters when using the RBF kernel (Eq.(3.19)). Corresponding to real-value coding, genetic operators are performed as follows:

- Crossover

$$X_1^{old} = \{x_{11}, x_{12}, .., x_{1n}\},\ X_2^{old} = \{x_{21}, x_{22}, .., x_{2n}\} \tag{3.23}$$

$$x_{1t}^{new} = x_{1t} + \sigma(x_{2t} - x_{1t}),\ t \in [p_1, p_2] \tag{3.24}$$

$$x_{2t}^{new} = x_{2t} - \sigma(x_{2t} - x_{1t}),\ t \in [p_1, p_2] \tag{3.25}$$

$$X_1^{new} = \{x_{11}, .., x_{1p_1-1}, x_{1p_1}^{new}, .., x_{1p_2}^{new}, x_{1p_2+1}, ..., x_{1n}\} \tag{3.26}$$

$$X_2^{new} = \{x_{21}, .., x_{2p_1-1}, x_{2p_1}^{new}, .., x_{2p_2}^{new}, x_{2p_2+1}, ..., x_{2n}\} \tag{3.27}$$

where p_1 and p_2 are two random values of cut points. X_1^{old} and X_2^{old} represent the pair of parents before crossover operation; X_1^{new} and X_2^{new} represent offspring. In addition, σ, which has the range of [-1,1], is a random micro number that controls the variance of each crossover operation. In other words, we partially perturb the parent in directions of the differential vector between two parents.

- Mutation

$$X^{old} = \{x_1, x_2, .., x_n\} \tag{3.28}$$

$$x_k^{new} = LB_k + \sigma(UB_k - LB_k) \tag{3.29}$$

$$X^{new} = \{x_1, x_2, .., x_k^{new}, .., x_n\} \tag{3.30}$$

  where k is the position of the mutation. LB and UB are the lower and upper bounds on the parameters. LB_k and UB_k denote the lower and upper bounds at location k. X^{old} and X^{new} represent the individuals before and after the mutation operation.

The stopping criteria are that the number of generations reaches 600 or the best fitness value does not improve during the last 100 generations. The best chromosome of the final generation is selected as the solution of the problem.

We use k-fold cross validation technique to assess the results. In this method, the original data is randomly partitioned into k equal sized sub-parts. The cross-validation process is repeated k times (the folds). At step k, the k^{th} part is used for testing by the trained model, which is built based on the remaining $k-1$ sub-parts. The k results from the folds can then be averaged (or otherwise combined) to produce a single estimation. The advantages of this technique are that all of the test sets are independent and the reliability of the the results could be improved. In our experiments, we choose $k = 10$ and combine k results to estimate the performance of the classifiers.

3.3.3 Results

Table 3.3 compares the performance of approaches in terms of the accuracy and F1 score. The best and second best values are marked in bold, and italic bold, respectively. For imbalanced data, the F1 score should be put in a higher priority than the accuracy. As can be seen, the proposed model achieves the best results on all datasets. It improves an accuracy of 2.26-7.98%, and an F1 score of 4.23-16.65% in comparison with the second best.

Weighting software metrics is beneficial to build predictive models. Indeed, very low values of F1 score indicate that most of the learning approaches have been suffered from imbalanced data. Especially, in the cases of Boosting, and SVM with grid search, the F1 scores are 0. This means that none of the defective modules is detected. Meanwhile, detecting defective modules is more important than that of clean modules. In contrast, the weighting model using GA and SVM can avoid bias in majority classes. It improves both measures of the accuracy and F1.

Table 3.3: Performance comparison in terms of Accuracy and F1-score

	CM1		jEdit1		jEdit2		KC1		KC2		KC3	
Method	Acc.	F1	Acc.	F1	Acc.	F1	Acc.	F1	Acc.	F1	Acc.	F1
C4.5	88.01	3.60	70.37	71.40	60.88	63.20	73.35	35.80	75.47	47.70	86.20	34.80
Random Forest	87.78	6.90	71.48	71.60	***66.12***	***69.00***	75.33	***38.40***	76.80	54.50	85.28	11.10
Naive Bayes	84.16	***30.00***	67.78	61.00	52.89	34.00	73.35	38.20	79.20	50.00	81.90	***37.90***
Boosting	***89.14***	0.00	70.37	68.00	56.47	56.40	74.34	15.30	***79.47***	***58.80***	85.28	25.00
SVM grid search	***89.14***	0.00	***76.30***	***75.90***	65.56	***69.00***	***76.73***	31.20	78.67	48.10	***87.12***	4.50
GA-SVM	**91.40**	**40.63**	**81.85**	**81.78**	**74.10**	**75.39**	**80.03**	**48.73**	**83.73**	**63.03**	**90.80**	**54.55**

The approaches are also verified according to the ability to distinguish among classes. This measure is estimated by the AUC (the area under the receiver operating characteristic (ROC) curve). Table 3.4 shows the comparison based on the AUC. Similar to the case of the F1, the SVM with grid search achieves low AUC values. Its performance is equivalent to a random classifier (having an AUC of 0.5). However, when combining with GA, the discrimination ability of the SVM is enhanced notably.

Table 3.4: Comparing the approaches according to the AUC measure

Method	CM1	jEdit1	jEdit2	KC1	KC2	KC3
C4.5	0.566	0.747	0.632	0.603	0.653	0.579
Random Forest	**0.751**	*0.804*	*0.734*	**0.687**	*0.767*	**0.763**
Naive Bayes	*0.7*	0.7	0.682	*0.669*	0.766	*0.74*
Boosting	0.671	0.789	0.606	0.661	**0.799**	0.737
SVM grid search	0.5	0.763	0.652	0.584	0.654	0.512
GA-SVM	0.632	**0.819**	**0.745**	0.659	0.733	0.701

Analyzing the behavior of algorithms is not based on the final results, but also on results during the execution process. To assess the combination ability between GA and SVM, we recorded the fitness value (accuracy) of the best solution every 10 generations. As shown in Fig. 3.5, the performance of SVM classifiers are improved continuously and quickly converge to the optimal solutions after around 50 generations on most of the experimental datasets. For KC3 and KC1 datasets, the rates of convergence are slower than others. The optimal solutions are correspondingly obtained after near 150 and 250 generations.

From above analysis, we can conclude that software metrics have different contributions to class labels and exploiting their roles is useful to build predictive models accurately. Our proposed model of GA-SVM is an efficient tool to optimize the model parameters and attribute weights simultaneously. It can improve the classification performance based on various measures like the accuracy, F1, and AUC.

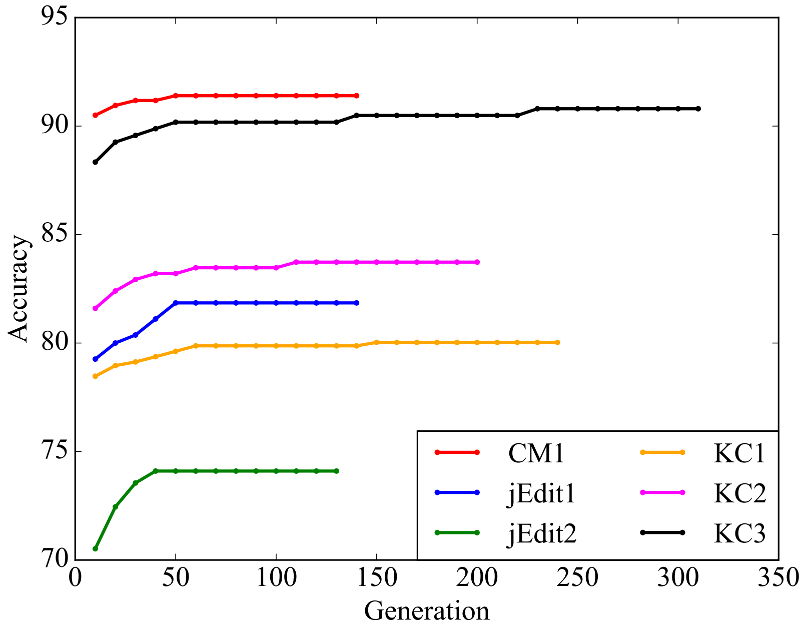


Figure 3.5: Running process of the GA-SVM model on the six datasets.

3.4 Related work

Various algorithms have been proposed in the literature of feature weighting. These algorithms can be divided into two groups: one which searches a set of weights through an iterative algorithm and uses the performance of the classifier as feedback to select a new set of weights [59, 73, 74]; the other computes the weights using the pre-existing bias model, e.g. conditional probabilities, class projection, and mutual information [21, 33, 68].

For the iterative approaches, Wu employed an evolutionary computation based method, namely Artificial Immune System (AIS), to find optimal attribute weight values automatically for weighted NB classification (AISWNB) [94] . The performance of the proposed method was validated on 36 UCI datasets and six image classification datasets from Corel Image repository. The experimental results demonstrate that the AISWNB method can significantly outperform its peers in classification accuracy, class probability estimation, and class ranking performance.

Lee proposed a new paradigm of weighting method, which assigns a different weight to values of each feature [51]. The method is called value weighting and implemented in the context of naive Bayes using wrapper method (VWNB). They reported that the VWNB method improves the performance of naive Bayes significantly and can be competitive with other state-of-the-art supervised algorithms.

Regarding the pre-existing bias model, Sáez focused on imputation methods to improve k-NN classification [76]. Under the imputation methods, the weight for each feature is estimated based on the rest of data and the Kolmogorov–Smirnov nonparametric statistical test is utilized to measure the changes between the original and imputed distribution of values. Three used imputation methods are k-NN Imputation (kNNI) [6], using Support Vector Machine to fill in missing values (SVMI) [26], Concept Most Common (CMC) [32]. They showed that their method was an effective way of improving the performance of the Nearest Neighbor classifier.

Jiang used a deep feature weighting (DFW) approach, which estimates the conditional probabilities of naive Bayes by computing feature weighted frequencies from training data [39]. Firstly, they applied the correlation-based feature selection (CFS) to select relevant features, and then defined weights of selected features and non-selected features as 2 and 1, respectively. These values are used to estimate the conditional probabilities of naive Bayes. The experiments on 36 UCI datasets show that their method rarely degrades the quality of the model compared to standard naive Bayes and, in many cases, improves it dramatically.

Xiang proposed a novel attribute weighting framework called Attribute Weighting with Smooth Kernel Density Estimation (AW-SKDE) [95]. In the AW-SKDE framework, the attributes weights are generated by calculating the mutual information between the features and the class label. They made an assumption that if one attribute shares more mutual information with the class label, that attribute will provide more classification ability than other attributes, and should therefore be assigned a higher weight. The experimental results showed that the AW-SKDE algorithm achieves comparable and sometimes better performance than the classical naive Bayes as well as other algorithms using a relaxed conditional independence assumption. However, their algorithm suffers from over-fitting.

Chapter 4

Tree-based Approaches

The previous chapter introduced traditional approaches based on hand-craft features, called software metrics. This chapter investigates more convenient approaches that automatically learn the features from programs' tree representations, known as abstract syntax trees (ASTs). We conduct experiments using various algorithms from common learning to deep learning to prove that ASTs are efficiently adapted to problems in source code analysis. This chapter provides the main contents as follows:

- Introduction to abstract syntax trees.
- Surveying different learning models on tree structures including tree edit distance (TED), tree-based convolutional neural networks.
- Presenting a technique for pruning redundant branches and reconstructing ASTs to boost classification performance in terms of accuracy and running time.
- Developing a sibling-subtree convolutional neural network (SibStCNN).
- Combining deep neural networks and common learning algorithms to make more powerful models.

These approaches are verified by a task of classifying source code according to functionalities.

4.1 Abstract Syntax Trees

In computer science, an abstract syntax tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language. Fig. 4.1 shows an example of the AST of the code for the Euclidean algorithm

Algorithm 1: The Euclidean algorithm

```
while $b \neq 0$ do
    if $a > b$ then
        $a = a - b$;
    else
        $b = b - a$;
    end
end
return a;
```

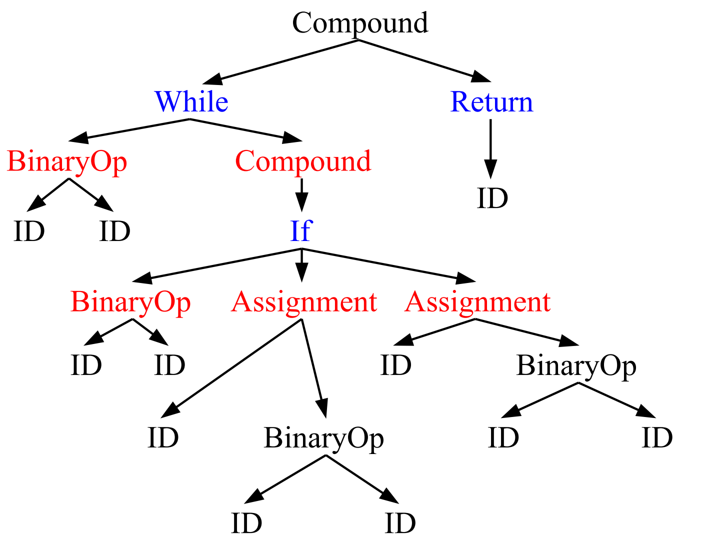


Figure 4.1: The AST for the code for the algorithm 1

Each node of the tree represents an abstract component occurring in the source code. An AST is a product of the syntax analysis phase of a compiler. It serves as an intermediate representation before generating code for the program. AST structures are widely used in compilers as well as programming language processing due to following advantages.

- Unlike source code, ASTs do not contain inessential elements such as braces, semi-colons, parentheses, and comments.
- The AST can be modified to optimize the program so that it executes more rapidly, or uses less memory storage or other resources. Some optimization techniques include replacing an expression with shorter/faster expressions, reordering of arithmetic operations or branches, and extracting common subexpressions (CSE). Meanwhile, such editing is impossible with the source code of a program.
- An AST contains extra information about the program. An example is the position of an element in the source code that may be used to notify the user of the location of an error in the source code.

4.2 Learning Approaches on Tree Structures

4.2.1 Tree Edit Distance (TED)

The tree edit distance (TED) of two trees is defined as the minimum cost sequence of node edit operations that transform one tree into another [9]. A rooted tree T is called as a labeled tree if each node is assigned a symbol from a fixed finite alphabet Σ; T is called an ordered tree if a left-to-right order among siblings in T is specified. Given an ordered labeled tree T, there are three basic tree edit operations as follows.

- Rename: Change the label of a node v in T.
- Delete: Remove a non-root node v in T with the parent v', in which case the children of v are promoted to be children of v'. The children are inserted in the place of v so that their relative order is retained.
- Insert: a node v is inserted as a child of v' in T. When inserting v, it becomes the parent of a consecutive sequence of the children of v'.

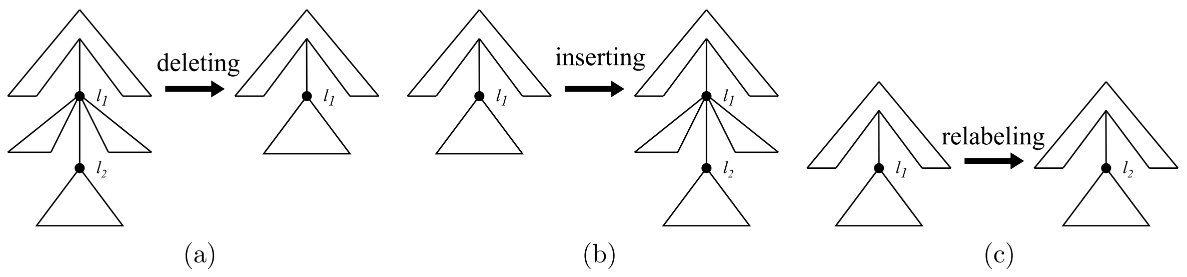


Figure 4.2: Basic tree edit operations. (4.2a) Deleting the node labeled l_2. (4.2b) Inserting a node labeled l_2 as the child of the node labeled l_1. (4.2c) A relabeling of the node label l_1 to l_2.

Fig.4.2 illustrates the basic edit operations for an ordered tree. Given trees T_1 and T_2, there exists many different sequences that transform from T_1 into T_2. Assume that we define a cost function on each edit operation. The cost of each sequence is the sum of the costs of its operations. Then, the tree edit distance (TED) between T_1 and T_2 is determined as the sequence with the minimal cost.

To calculate TED, various algorithms have been proposed and improved efficiency in terms of computational time and memory requirements [20,69,70]. In this work, we apply a robust and memory-efficient algorithm for the tree edit distance namely AP-TED (All Path Tree Edit Distance) [70] to compute the distance between AST trees.

4.2.2 Tree-based Convolutional Neural Networks (TBCNN)

Tree-based convolutional neural network (TBCNN) is a novel model proposed by Mou [66], which showed a notable performance on program classification problem. Fig.4.3 illustrates the architecture of the TBCNN. Firstly, each AST node is represented as a vector by using a coding layer. The task of this layer is to embed AST symbols in a continuous vector space where semantically similar symbols are mapped to nearby points. For examples, the symbols `While` and `For` are similar because they are loop statements. But they are different from `ID` which may present some data.

After coding, each node in ASTs is represented as a real-value vector $x \in \mathbb{R}^{N_f}$. Then the author designed a set of fixed-depth subtree detectors sliding over entire AST to

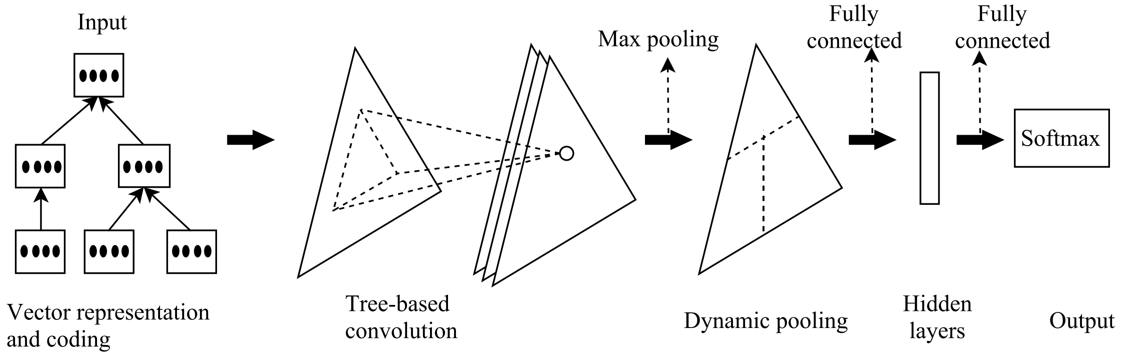


Figure 4.3: The architecture of the tree-based convolutional neural network (TBCNN).

extract structural information of the program. The output of the the feature detectors is computed by the following equation.

$$y = \tanh(\sum_{i=1}^{n} W_{conv,i} \cdot x_i + b_{conv}) \tag{4.1}$$

where, $x_1, .., x_n$ are vector representations of nodes inside the sliding window, $y, b_{conv} \in \mathbb{R}^{N_c}, W_{conv,i} \in \mathbb{R}^{N_c \times N_f}$. ($N_c$ is the number of feature detectors.).

One problem is that determining the number of weight matrices in Eq 5.2 is impossible because AST nodes have various numbers of children. To solve the problem, the authors proposed the notation of "continuous binary trees" whereby the convolutional layer only uses three weight matrices as parameters including W^t_{conv}, W^l_{conv}, and W^r_{conv} (superscripts t, r, l refer to "top", "left", "right"); the weight matrix for any node x_i is a linear combination of W^t_{conv}, W^l_{conv}, and W^r_{conv}, with coefficients η^t_i, η^l_i, and η^r_i, respectively. The coefficients are computed based on the relative position of x_i in the sliding window as following equations:

- $\eta^t_i = \frac{d_i - 1}{d - 1}$ (d_i: the depth of the node i in the sliding window; d: the depth of the window.)
- $\eta^r_i = (1 - \eta^t_i)\frac{p_i - 1}{n - 1}$ (p_i: the position of the node; n: the total number of p's siblings.)
- $\eta^l_i = (1 - \eta^t_i)(1 - \eta^r_i)$

The pooling layer thereafter is stacked to gather the extracted features over parts of the tree. To produce a fixed-sized output from variable-sized ASTs, the authors apply two ways of dynamic pooling called one-way pooling and three-way pooling [82]. According to the experimental results, the performance of the two pooling methods is similar. Finally, a fully connected layer and an output layer are added for supervised classification.

4.2.3 Sibling-subtree Convolutional Neural Networks (SibStCNN)

In the convolutional layer of SibStCNN, feature detectors are applied to exploit information both in depth and width dimensions from different parts of trees. Unlike TBCNN, the

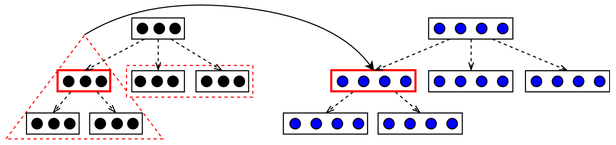


Figure 4.4: Sibling-subtree convolution. Nodes on the left are feature vectors of AST nodes. Nodes on the right are feature maps.

local regions for feature extraction of SibStCNN are expanded to the siblings. In ASTs, sibling nodes have the similar roles, and their information is relevant. Considering the subtree of `If` in Fig. 4.1, the node of the binary operator `a>b` has two children `ID:a` and `ID:b`, and the value of `a>b` is determined based on both values of a and b. Additionally, the subtree `If` includes three branches of the binary operator and two assignment statements, in which either of the assignment statements is active depending on the value of the operator. For these reasons, it is more precise when a node is evaluated based on information of itself and the surrounding nodes (descendants and siblings).

In this work, the subtree depth is set to 2; the number and widths of rectangles are varying according to the location of the sliding window and the number of siblings such that the window covers all children, siblings, and the current node (Fig. 4.4). Formally, in each position, if the current node (x_c^0) has C children and S siblings with the corresponding vector representations $x_c^i, i \in \{1,..,C\}$ and $x_s^j, j \in \{1,..,S\}$, then the feature maps are obtained as follows:

$$y = \tanh(\sum_{i=0}^{C} W_{conv_c^i} \cdot x_c^i + \sum_{j=1}^{S} W_{conv_s^j} \cdot x_s^j + b_{conv}) \tag{4.2}$$

An obstacle to computing the feature maps (Eq. 4.2) is that determining the number of weight matrices is unfeasible because AST nodes have different numbers of children, and siblings as well. To solve this problem, we use three weight matrices (W_{conv}^t, W_{conv}^l, and W_{conv}^r) for the subtree part and one weight matrix (W_{conv}^s) for the rectangle parts. In this scenario, $W_{conv_c^i}$ is a linear combination of W_{conv}^t, W_{conv}^l, and W_{conv}^r, with coefficients η_i^t, η_i^l, and η_i^r (Section 4.2.2); $W_{conv_s^j}$ is a scale of W_{conv}^s with the ratio $\frac{c_{s_j}}{s_{s_j}}$(c_{s_j}: the number of s_j's children; s_{s_j}: the number of s_j's siblings).

4.3 Combinations of Deep Neural Networks and Traditional Learning

4.3.1 The Combination Model of kNN-TED and Tree-based Networks

The tree-based convolution kernels explore the information contained inside the AST nodes regardless of shapes and sizes of the trees, while TED measures the similarity between tree structures. In other words, these methods extract two types of information of ASTs. Thus, the cooperation between them provides stronger proof for the classifier to determine the label of an unknown instance.

For above reasons, we design a combination model of kNN-TED and TBCNN, in which the decision values for each unseen instance is estimated by Eq.(4.3) as follows:

$$DecVal_j^i = (1-t) * Prob_j^i + t * MF(nnDist_j^i) \tag{4.3}$$

where $DecVal_j^i$ is the decision value of instance i belonging to class j; $Prob_j^i$ is the prediction probability (produced by TBCNN) for class j of instance i; $nnDist_j^i$ is the sum of normalized distances between instance i with instances of class j in the set of k neighbors of i; MF is the mapping function, which transforms the value of $nnDist_j^i$ to [0,1]; t is the combination factor in the range of [0,1].

After that, the label of the instance is determined by Eq.(4.4):

$$L^i = \begin{cases} L_{n_1} & \text{if } L_{n_1} = L_{n_2} = .. = L_{n_k} \\ l & \text{if } DecVal_l^i = max\{DecVal_j^i\} \end{cases} \tag{4.4}$$

where L^i is the predicted label of instance i; $L_{n_1}, L_{n_2}, .., L_{n_k}$ are the labels of k neighbors of instance i.

To ensure the balanced contributions of kNN-TED and TBCNN to the combination model, the value of $nnDist_j^i$ is mapped from the range [0, $MaxDist$] to [0, 1] by the function MF. In our implementation, we use the training set and the validation set to build the best classification model; and, the test set is used to verify the performance of the model. To build the combination model, we have to find its parameters including the factor t and $MaxDist$. Firstly, TED is applied to compute all values $nnDist_j^i$ in the validation set. Next, the maximum of such values is assigned to the $MaxDist$. We train the TBCNN through 60 rounds and select the classifier which obtains the highest accuracy on the validation set. The parameter t is selected by tuning its values from 0 to 1 by step 0.02 such that the combination model estimated according to Eq.(4.3) and Eq.(4.4) achieves the best performance on the validation test. After acquiring the parameters t and $MaxDist$, the decision values for each unseen sample are easily computed based on Eq.(4.3); and the predicted label is determined according to Eq.(4.4).

4.3.2 The Integration Model of Tree-based Networks and SVM

To make more accurate prediction models, we integrate TBCNN/SibStCNN and an SVM classifier. In the models, the tree-based CNNs are able to capture underlying meaning inside tree nodes; the SVM classifier is a powerful classifier, which has shown state-of-the-art performance in a wide range of applications such as text categorization, hand-written character recognition, image classification, biosequences analysis and so on [7, 27, 54].

Fig. 4.5 illustrates the architecture of the integration model. It consists of two components: 1) a tree-based CNN extracts semantic features of programs, 2) SVM is adopted to build the classifiers based on extracted features. Specially, TBCNN/SibStCNN serves as a supervised approach to learn program vector representations from source code. We

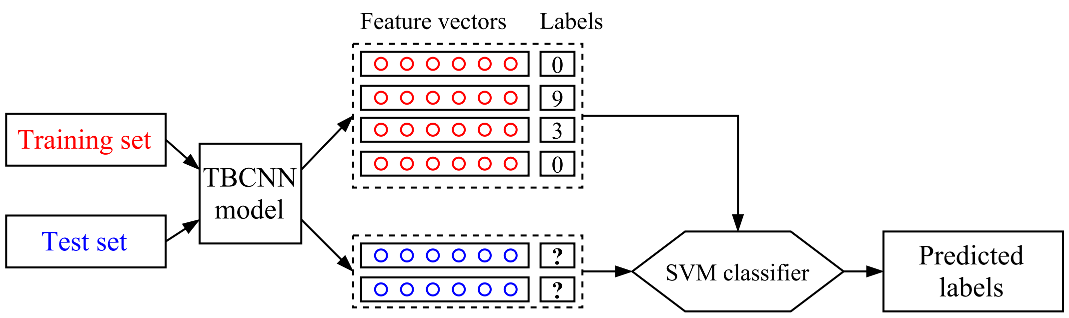


Figure 4.5: The integration model of tree-based CNNs and SVM for program classification.

keep the training procedure of the network. The best network model obtained from training process is applied to generate vector representations for all programs by getting the output signals of the final hidden layer (Fig.4.3). After that, an SVM classifier is built based on the vector representations of training instances. The process of predicting the class label of a program contains following steps:

- Parsing the source code into an abstract syntax tree.
- Feeding the TBCNN/SibStCNN models with the tree to generate the vector representation.
- Using the SVM classifier to predict the class label of the vector.

4.4 Experiments

4.4.1 Data Preprocessing

As mentioned above, the AST data is high-dimensional and need to be refined. Thus, we present a heuristic technique to prune redundant branches and reconstruct sub-tree structures based on observations on source code. The details of this technique are described as follows:

1. Eliminate structures of variables, constants, procedures, enumerations declaration and type definitions. In a programming language, the declaration statements are used to specify the data type (for variables and constants), or the type signature (for procedures). However, these properties of an identifier will be revealed in next statements, which manipulate such identifier. In other words, pruning the branches for these statements does not lead to a decrease in an amount of information on ASTs; and it removes unused identifiers. To illustrate, we analyze two below programs in files 46.txt and 84.txt in group 86.

Program 1 (file 46.txt):

```
int main()
{
 int n,i,b[100],j,t,m;
 scanf("%d",&m);
 for(int l=0;l<m;l++)
{
  scanf("%d",&n);
  int *a=(int*)malloc(sizeof(int)*(n+2));
  for(i=0; i<n; i++)
   {
      scanf("%d",&a[i]);
   }
      .......
}
return 0;
}
```

Program 2 (file 84.txt):

```
int main()
{
 int n,i,a[100],b[100],j,t,m;
 scanf("%d",&m);
 for(int l=0;l<m;l++)
 {
      scanf("%d",&n);

     for(i=0; i<n; i++)
     {
        scanf("%d",&a[i]);
     }
     .......
 }
 return 0;
}
```

The differences between two programs are the declaration locations and data types of variable `a`. The function of variable `a` in both programs is to represent a set of values. In program 1, `a` is an array; In program 2, `a` is a pointer. In terms of function, two programs are similar and they always produce the same output with the same input data. However, the parse trees of the programs are different. If we eliminate the variable declaration branches, the parse trees of two program are the same. The other notice is that the roles of variables can be revealed via operators since they are manipulated. For example, `a` is known as a set of values due to the use of operator `[]`; `l` is known as a number when it is assigned to `0`.

2. Cut down two branches of trees of `For` statements. The ASTs for `For` statements have four children representing different parts inside them (Fig.4.6a) including `Init` (initialization), `Condition` (termination expression), `Next` (increment expression), `Statement` (body). When designing a `For` statement, programmers usually put the main works into the body section. The termination expression is responsible for controlling `For` loops. Whereas `Init` and `Next` sections contain little information about the tasks of `For` statements.

   Indeed, after observing programs in the dataset, we found many `For` statements are written in the simple form. The following snippet code is an example.

```
for(;x[k]==y[j]&&k>=0&&j>=0;)
   {
     k--;
     j--;
   }
```

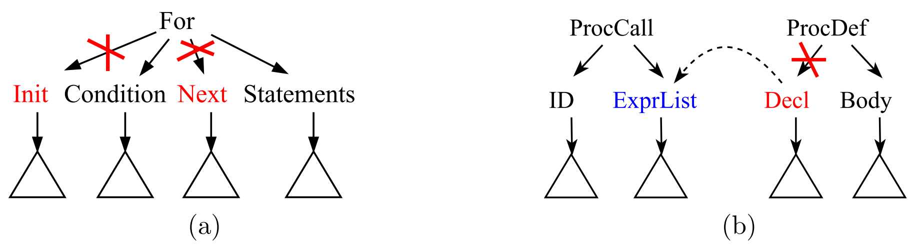


Figure 4.6: The operations of pruning redundant branches of `For` and `ProcDef` (Procedure Definition) sub-trees.

For above reasons, we remove two uninformative children of ASTs of `For` statements including `Init` and `Next` (Fig.4.6a).

3. Cut down the declaration branch of trees of procedure definition statements. The ASTs of procedure definitions have two main branches including `Decl` (Declaration) and `Body`, where the `Decl` contains return type, procedure name, and parameter list sections. The parameter list section defines the temporary variables used in the procedure. When the procedure is invoked, the actual parameters are passed. It means that there exists duplicate information in the program because the information of temporary and actual parameters is similar. Therefore, we eliminate `Decl` branches of procedure definition ASTs (Fig.4.6b).

4. Rename the root nodes of trees of `For`, `While`, and `Do-While` statements. After cutting down redundant branches, the ASTs of `For`, `While`, and `Do-While` statements have the same structure, which involves two children namely `Condition` and `Statement`. To reduce the symbols of AST nodes, we change the AST node names from `For`, `While`, and `Do-While` into `Loop`. Using fewer symbols is beneficial because it reduces the complexity of the dataset.

4.4.2 The Dataset

The proposed approaches are verified by a task of program classification in which programs performing the similar tasks are assigned to the same group. The dataset is obtained from a pedagogical programming open judge (OJ) system shared by Mou [66]. It contains programs for 104 programming problems (considered as target labels) in which each of them includes 500 programs. Programs with the same target label have the same functionality. The dataset was split by 3:1:1 for training, validation, and testing.

Table 4.1 shows statistics on the AST datasets. The statistical figures indicate the challenges of working with AST data due to their shapes and sizes are very large and different. For the original ASTs, the numbers of tree nodes are varying from 29 to 7027; the average number of nodes is 189.6; the standard deviation is 106. Thus, preprocessing ASTs to reduce its complexity and noisy data as well is essential. It is noted that depending on each specific problem, we must select appropriate refining methods to avoid losing so much

Table 4.1: Statistics of the dataset (ASTs$_{OR}$ are the original ASTs, ASTs$_{MP}$ are the ASTs after pruning minor procedure branches, ASTs$_{HP}$ are the ASTs after applying heuristic pruning).

Statistics	ASTs$_{OR}$		ASTs$_{HP}$		ASTs$_{MP}$	
	Mean	Std.	Mean	Std.	Mean	Std.
# of AST nodes	189.6	106.0	134.0	92.1		
# of AST leaves	90.5	53.8	66.5	47.7		
Avg. leaf nodes' depth in an AST	7.6	1.7	8.3	1.9		
# nodes of the smallest AST	29	-	7	-		
# nodes of the largest AST	7,027	-	6,999	-		

meaningful information. For example, regarding classifying programs by functionalities, considering the similar roles of `For`, `While`, and `DoWhile` statements is compatible with this problem. From Table 4.1, applying the heuristic pruning method reduces the node numbers notably and increases the average leaf nodes' depth in ASTs. This means most of the redundant branches are shorter than the meaningful branches.

4.4.3 Experimental Setup

To address the program classification problem, we converted programs into ASTs and then surveyed various algorithms, which treat the ASTs as the input data. The details and settings of these algorithms are described as follows.

TBCNN, SibStCNN. Initial learning rate is 0.3; vector dimension is 30; convolution layers' dimension and penultimate layers' dimension are the same value 600; the loop iteration is 60; the activation function of the output layer is softmax.

The k-Nearest Neighbor (kNN) + TED, Levenshtein distance(LD). To employ the kNN algorithm, the distance between programs is estimated based on their ASTs. We used two methods to compute the distance including TED [70], and Levenshtein distance [36]. The Levenshtein distance (LD) is a measure of similarity between two sequences. The ASTs are traversed to generate sequence representations for programs. The number of nearest neighbors is set to 3.

TBCNN + kNN-TED. The settings of TBCNN is kept. The number of nearest neighbors is expanded to 10 with the aim of providing the combination model with more proof to make the final decision on instance labels. The mapping function is applied in the same manner for both validation and test sets.

TBCNN + SVM. The settings and the training procedure of TBCNN are kept. The dimension of output vectors is 600. The hyper-parameter C of SVM classifiers is chosen from $\{1, 2, 3\}$. We build the classifiers with different kernels including linear, polynomial, radial basis function (RBF), and sigmoid.

Tree kernel SVM. We used SVM-light ([65]) and combined both tree kernels and feature vectors of BOT. SVM-light is designed for binary classification problems. To adapt for multiclass classification problems, we used *one-vs.-all* strategy, whereby a single classifier is trained for each class, with the samples of that class as positives and all another samples as negatives. In other words, for K-label problems, we must train K classifiers and use all of them for making a decision about the label of each sample. In the prediction stage, the label of an unseen instance corresponds to the classifier which produces the highest confidence score. The setting for the SVM-light includes: the kernel is the combination of forest and vector set; kernel to be used with vectors is chosen from linear, polynomial, radial basis function (RBF), and sigmoid tanh; decay factor in tree kernels is 0.4; The normalization is applied to each individual tree and vector. The dataset contains 104 target labels and the instances of each class are 500. To avoid facing with imbalanced data when training classifiers using *one-vs.all* method, we used the sub-sampling technique to reduce the negative instances so that the proportion of positives to negatives is 1:20.

Gated Recurrent Neural Network (GRNN). The GRNN is successful in classifying documents [87]. To adapt this model for program classification, each program is considered as a document, whereby each statement is equivalent to a sentence. The documents are generated by traversing all subtrees of statements in the ASTs using depth-first search algorithm. In the experiments, the vector representation for AST symbols is learned by using a word2vec model. The vector size is set to 30.

4.4.4 Results and Discussion

Table 4.2 shows the performance of classifiers in terms of accuracy and running time. For the case of without pruning, TBCNN yields a remarkable accuracy of 92.63% due to subtree feature detectors which have good ability to capture underlying meanings of AST nodes. Thanks to the expansion of sliding windows, SibStCNN improves the accuracy of TBCNN by 0.63%. The accuracies of the combination models between SibStCNN/TBCNN and kNN indicate that learning trees from many perspectives is beneficial. SibStCNN/TBCNN and kNN outperform other methods, achieving the highest accuracies of 94.13% and 93.48%, respectively.

It is worth noticing that, the tree-based algorithms outperform the sequence-based algorithms. Although kNN is one of the simplest machine learning algorithms, it yields higher accuracy than that of GRNN - a deep neural network which works with sequences. The main reason is that ASTs contain rich and explicit information about programs. Meanwhile, the position information will be lost when converting ASTs into sequences by traversing. The loss of information leads to a decrease in the strength of classifiers. Additionally, we extracted 17 file metrics from programs and used Weka's implementation of different algorithms including support vector machines (SVMs), naive Bayes, and kNN to classify the metrics data. However, the accuracies of these classifiers are very low, around 22%.

We tried tree kernels on AST structures and achieved the worst accuracy in comparison with the other methods. We observed that the tree kernels could not analyze deeply the semantic meanings of ASTs. They only captured the popular subtrees, which occur in almost ASTs and may not be relevant to the main function of programs. An example of a structure extracted by tree kernels is "`Decl(TypeDecl(IdentifierType int)`". This

Table 4.2: Performance comparison of the pruning approaches in terms of accuracy and execution time (ASTs$_{OR}$ are the original ASTs, and ASTs$_{HP}$ are the ASTs after heuristic pruning)

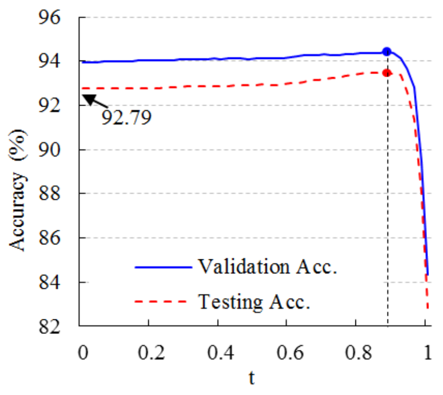
Method	ASTs$_{OR}$ Acc. (%)	ASTs$_{OR}$ Avg. time (s)	ASTs$_{HP}$ Acc. (%)	ASTs$_{HP}$ Avg. time (s)
kNN + TED	85.84	259.59	**86.35**	108.8
kNN + LD	83.08	4.78	**85.56**	2.39
Tree kernel SVM	58.39	0.26	**62.65**	0.249
GRNN+LSTM	80.61	454.37	**83.31**	338.2
TBCNN	92.63	1194	**92.88**	810
SibStCNN	**93.26**	1494	93.14	944
TBCNN+ kNN-TED	93.48	-	**93.69**	-
SibStCNN+ kNN-TED	**94.13**	-	93.98	-

structure is the AST of a declaration statement of a variable of type integer such as "`int a;`". In programming languages, to determine the function of a program, we must consider more complex structures inside it. For instance, AST structures of `For`, `While` statements should be similar since they are related to control flow; and, they are different from `Constant` because a `Constant` represents an unmodified value.

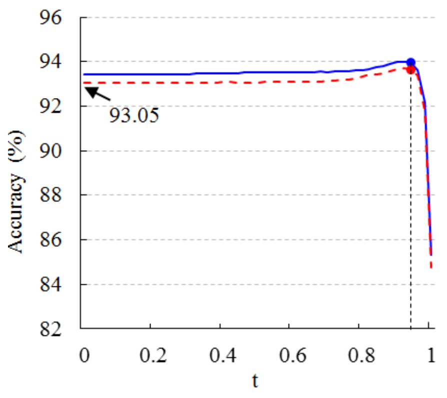
For AST data structures, dimension reduction is an essential task due to large shapes and sizes of the trees. To reduce the complexity of the data and maintain the semantic meaning of ASTs, we propose a technique to prune redundant branches and reconstruct sub-trees. Table 4.2 compares the performance of classifiers in terms of accuracy and computational time in cases of before and after pruning trees. Where the average computational time of the algorithms is estimated as follows: TBCNN and LSTM + GRNN are the running time each of loop iteration; the others are the time to predict an instance.

The results in Table 4.2 show high efficiency of the pre-processing data techniques. For the heuristic pruning, it not only enhances the accuracies of all classifiers but also reduces the execution time notably. The execution time of kNN-TED and kNN-LD decreases more than two times; the execution time of TBCNN and GRNN decreases nearly 1.5 times; the execution time of SVM-Tree kernel decreases slightly. These prove that pruning redundant branches can efficiently eliminate noisy information without a loss of useful information.

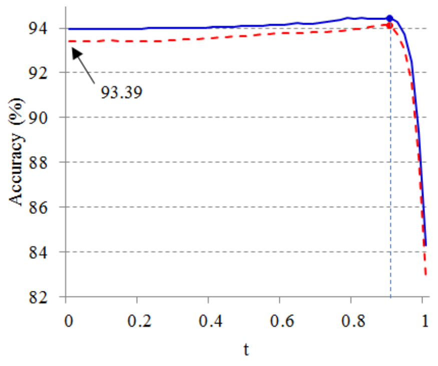
It is interesting that due to the application of the heuristic pruning techniques we detected 356 duplicate instances, which may not be found when using the original trees. In other words, many students copied completely or copied with tiny modifications the solutions from the others and used such solutions to submit to the OJ system. For example, the contents of file 2557.txt in group 62 and file 892.txt in group 26 completely overlap; the programs in files 46.txt and 84.txt in group 86 are similar. The differences between them only include the position of variable declaration statements and the use of a pointer instead of an array to represent a set of numbers. These prove that pruning and reconstructing tree approaches extract main contents which show the major tasks of the programs; and these approaches provide a feasible solution to solve source code clone detection problem in the area of software engineering.



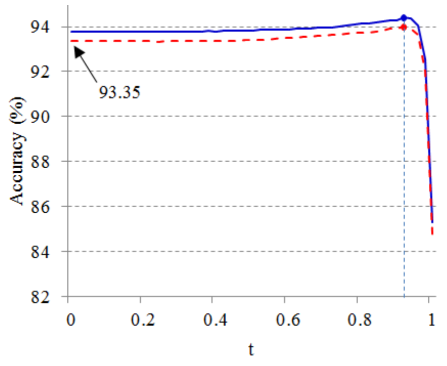
(a)



(b)



(c)



(d)

Figure 4.7: Tuning the factor t for the combination models of TBCNN/SibStCNN and kNN-TED. (4.7a) and (4.7c) before pruning trees, (4.7b) and (4.7d) after heuristic pruning.

To take advantage of different types of ASTs' information, we proposed the hybrid model of TBCNN and kNN, where the task of TBCNN is extracting the underlying meanings inside tree nodes and kNN measures the differences between tree structures. In our implementation, we find the best parameters for the combination model by using training and development sets. Fig.4.7 illustrates the process of tuning the combination factor t in cases of before and after pruning trees. As can be seen, the two lines representing the change of accuracies on the validation set and the test set according to t have the same trend. The accuracies increase when t is adjusted from 0 to around 0.9; after peaking the top at t around 0.9, the accuracies begin to drop down to the kNN classifier accuracies. From Eq(4.3), t is the trade-off parameter between TBCNN and kNN of contribution to the model. The down arcs are caused by the bigger contribution of kNN when t increases. Especially, all sub-figures in Fig.4.7 show that at $t = 0$, the accuracies of the hybrid model are higher than those of TBCNN (Table 4.2). According to the Eq. 4.4, the label of an unseen instance is predicted based on kNN if ten neighbors are in the same group, otherwise, the label is the output of TBCNN/SibStCNN. These mean that although many structures are similar, the tree-based CNNs could not detect them. Thus, using extra structural information of ASTs is helpful in making the final decision on class labels of instances.

The SVM classifier is known as a powerful classifier, which has high accuracy and ability to deal with high-dimensional data. Tree-based CNNs are able to capture underlying meaning inside tree nodes. For this reason, we generated the integration models of the tree-based CNNs and SVM. Table 4.3 compares our proposed models with the tree-based CNNs. Generally, the proposed model improves the accuracy of TBCNN/SibStCNN remarkably. The SVM with RBF kernel outperforms others on all experimental datasets.

Table 4.3: Accuracy of the combination models of TBCNN/SibStCNN with SVM.

Method		ASTs$_{OR}$	ASTs$_{HP}$
TBCNN		92.63	92.88
	Kernel		
	Linear	93.42	93.43
TBCNN+SVM	Polynomial	91.69	91.58
	RBF	**93.74**	**93.89**
	Sigmoid	93.63	93.60
SibStCNN		93.26	93.14
	Kernel		
	Linear	93.74	93.60
SibStCNN+SVM	Polynomial	91.93	91.43
	RBF	**94.24**	**94.02**
	Sigmoid	94.15	93.90

For machine learning approaches, the data dimensions are subject to the performance of algorithms regarding computer memory, computational time and the accuracy. Specifically, high-dimensional data lead to a waste of computer memory and computational time, while too few attributes may weaken algorithmic efficiency because the information about instances is not provided sufficiently. To verify the effect of attribute quantity to classifiers, we ran the TBCNN and TBCNN-SVM models with varying numbers of hidden nodes. Fig. 4.8 shows that the computational time of the SVM increases rapidly with the number of the hidden nodes. Besides, the accuracies of both models on validation and test sets are high and stable when the hidden nodes are greater than 100. All accuracy curves fall down sharply when the hidden nodes turn from 100 to 25. These proofs enable us to conclude that choosing a proper number of hidden nodes or the number of features is vital for neural networks and SVMs. In the experimental dataset, if the accuracy is more interesting, the number of hidden nodes should be 200; this number is 100 in case we concern both about the accuracy and the running time.

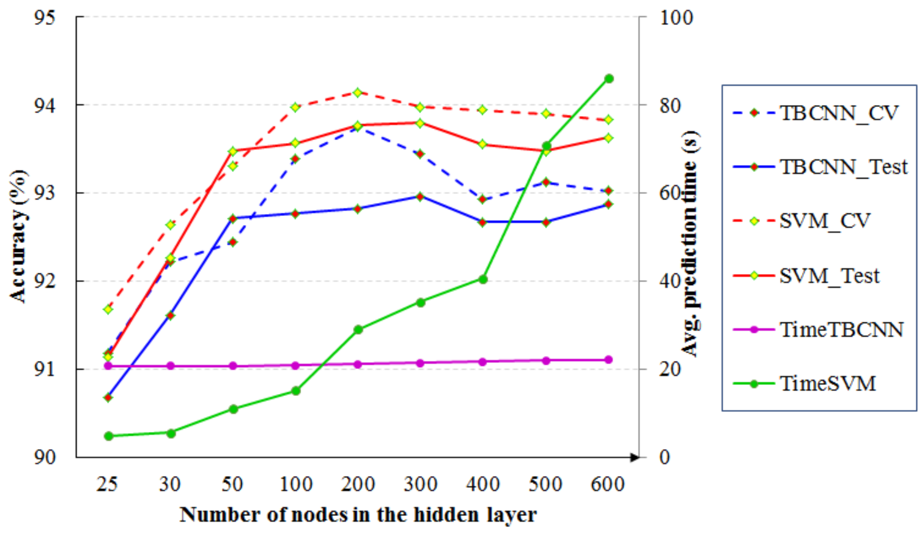


Figure 4.8: The accuracies on validation and test sets in the case of heuristic pruning (HP), and the time for predicting an instance of classifiers with different numbers of hidden nodes. TBCNN_CV, TBCNN_Test, SVM_CV and SVM_test show the accuracies of TBCNN and TBCNN-SVM classifiers on validation and test sets, respectively. TimeTBCNN and timeSVM represent the time for predicting an instance using TBCNN and SVM.

4.4.5 Data Analysis

In this section, we present some observations on the experimental dataset and the output of the classifiers as well. For the TBCNN model, the feature detectors ignore the shape and size of an AST when collecting information of the tree. The extracted vector of each node is computed from its descendant vectors inside the window of the feature detector. Thus, given an AST, if we move a branch of a node to be a child of other node, the extracted information of these nodes may be changed. In programming language C/C++, we can change the location of a statement without effects on the program execution. For example, a variable declaration statement can be placed in any position above the location where the variable is used. Moreover, to implement a task, we can select different statements such as `print` or `cout` for output stream; `for`, `while`, or `do...while` for iterative control flow. Due to the flexible design of programs, although their ASTs have different appearance, they may perform the same task. When observing the output of TBCNN and kNN, we saw that kNN classifier was able to provide many similar structures for predicted instances, while TBCNN failed to capture information of these instances due to above changes. Therefore, to create a more powerful predictor, the output of kNN is used to assist the TBCNN in making final decisions.

When working with AST data, many redundant information needs to be eliminated. However, we must find a suitable method such that it avoid loss of meaningful nodes. After pruning redundant branches, we saw that all algorithms work more accurately due to avoidance of ambiguity in various cases such as using a pointer or an array to represent a set of values; changing the locations of variable, procedure declarations; the use of `for`, `while` and `do...while` to control a flow.

4.5 Related Work

Source code analysis has been widely applied to a variety of software engineering tasks such as clone detection [8, 42], fault location [41, 61], quality assessment [43, 79], and so on. The advantages of these approaches are providing much predicted information about new products based on other applications. Such information is very helpful for enhancing software quality by avoiding the defects and reusing the resources of previous projects. Due to great benefits that solving software engineering problems brings to the software industry, numerous algorithms and techniques have been proposed and improved to make the predicting systems applicable in practice.

Ugurel et al. [88] applied machine learning approaches to automatically classify open source code into eleven application topics and ten programming languages. Firstly, feature extractors are utilized to generate the vector representation for each program/source code file. Then the SVM classifiers are trained on such feature vectors. Similarly, Alvares et al. [4] built source code classifiers by using lexical analysis, scoring strategies and an evolutionary algorithm. The task of the evolution algorithm is to filter the set of keywords for each programming language to strengthen the lexical-based classification analysis. The experiments on the real-world source code of more than 200 different open source projects show that the proposed approach can create high-performance source code classifiers. In order to improve source code quality, Lerthathairat et al. [52] proposed an approach that classifies source code with software metrics and fuzzy logic and then improves bad smell, ambiguous code. Chandra et al. [15] developed a tool based on CK metrics to predict the decomposition point of the class.

Despite the effectiveness of software metrics on specific problems, it is labor intensive and unable to extract patterns from raw data [30]. Therefore, many studies have aimed to automatically learn data features from graph or tree representations of source code by leveraging deep neural networks. For graph-based approaches, Binkley et al. [10] presented a collection of techniques which employ graphs as internal representations to improve source code analysis tools. Komondoor et al. [49] designed a tool that analyzes program dependence graphs (PDGs) and program slicing to find duplicated code and displays them to programmers. The tool is useful for refining source code to make it more well-organized by detecting and replacing all the clones by calls to the new procedures. Liu et al. [56] conducted experiments on mining PDGs to prove the efficiency and the effectiveness of graph-based approaches to plagiarism detection in programs having thousands of lines of code.

Recently, due to containing rich information about programs, the tree-based approaches have shown a notable success in dealing with obstacles of software engineering area. Wang et al. [92] applied a deep belief network (DBN) to automatically learn semantic features of programs for defect prediction. The experiments were conducted on various Java open source projects obtained from PROMISE repository[1]. The results indicate that the tree-based approaches significantly outperform metrics-based approaches in terms of precision, recall, and f-measure. Mou et al. [66] proposed a tree-based convolutional neural network (TBCNN) which work on tree structures with varying shapes and sizes. The model achieves very high performance for two software engineering tasks: classifying programs by functionalities and detecting bubble sort.

We would like to apply the TBCNN model for software engineering problems. We propose some data refining techniques and two combination models to enhance the accuracy of classifiers. Our methods firstly are verified on the same problems and dataset of Mou's work. In ongoing work, we adapt these approaches to the software defect prediction problem. For predicting bugs, errors, or faults in a source code, the smaller size of the source code, the more valuable the predictor is, because locating and fixing bugs are easier. Thus, the sizes of ASTs are not huge; and the model is completely applicable. In this chapter, we present two combination models of TBCNN and kNN-TED, SVM to solve program classification, in which tree structures serve as the input data. In addition, we proposed the pruning tree techniques to refine the data of ASTs. The experimental results show a significant improvement of classifier performance in terms of accuracy as well as execution time.

These models are applicable to other problems in the field of software engineering such as software defect prediction, and clone detection. The tree-based approaches may provide promising results because of following reasons:

- We can obtain ASTs from source code of programming languages with grammars by using a corresponding parser.
- For two above problems, we should process a part of programs because the smaller code snippet we predict the issues, the easier we locate and fix them. Therefore, the AST sizes are not too large.
- Software metrics are usually applied for big projects, and they are failed to capture the meaning of small source code (Section 4.4.4).

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

Chapter 5

Deep Neural Networks on Assembly Code

The previous chapter presented building models based on ASTs. However, AST just represents the structures and do not reveal the behavior of programs. To deeply explore into semantic meanings, this chapter formulates an end-to-end approach to program analysis. Regarding this, assembly instruction sequences serve as input data for learners instead of software metrics or abstract syntax trees. To enrich the formation, the instructions are viewed by various perspectives before feeding to machine learning algorithms. The main contents provided in this chapter are:

- Formulating an end-to-end approach that applies deep neural networks on assembly code.
- Presenting an algorithm for constructing control flow graphs of programs from assembly code.
- Designing two multi-view convolutional neural networks on assembly instruction sequences and control flow graphs.

The proposed approaches are verified based on two tasks: software defect prediction and malware analysis.

5.1 Assembly code and two views of data

Assembly code consists a series of low-level machine instructions that are close to machine code instructions. Assembly code can be converted into machine code by an assembler, and reversed by a disassembler. Transforming from a source file written in a programming language into assembly code is done by a corresponding compiler. Fig. 5.1 describes the process of compiling C source files into executable files. Assembly files are the products after the compiler analyzes and translates ASTs.

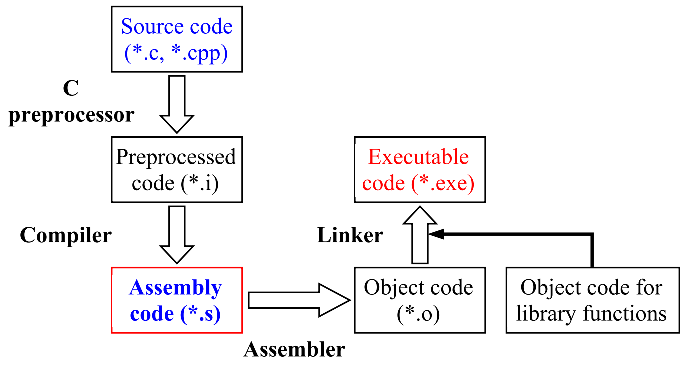


Figure 5.1: The process of compiling C programs from source code to executable files.

An assembly instruction is equivalent to a CPU-level instruction and usually involves an operation code mnemonic followed by a list of operands. For example, the instruction of `move ax, 128` guides the computer to copy the number 128 to the register ax.

To enrich data for learning, we use two views of data including instruction names and instruction groups. For instance, instructions `jne`, `jle`, and `jge` are tagged to the same group since they are conditional jump instructions. Similarly, `addb`, `addl`, and `addw` belong to the group of arithmetic instructions.

5.2 Convolutional Neural Networks on Instruction Sequences

We design a multi-view convolutional neural network on assembly instruction sequences, called ASCNN. The network is built 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:

$$c_i^f = f(W^f \cdot x_{i:i+h-1} + b^f) \tag{5.1}$$

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.

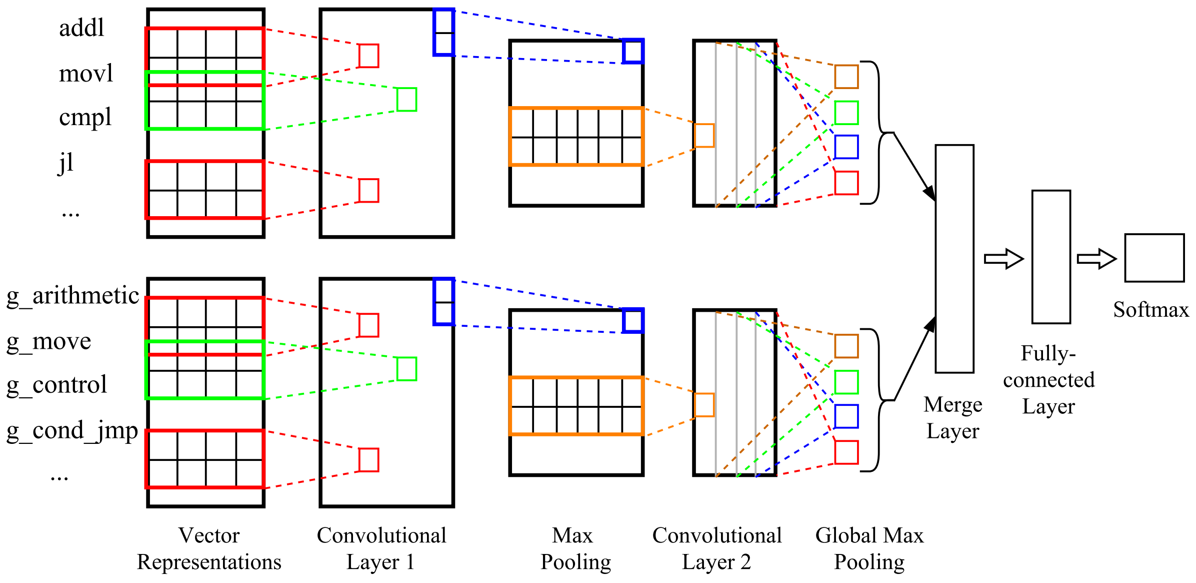


Figure 5.2: A convolutional neural network with two views for assembly instruction sequences.

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 [50]. 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. 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 [17].

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

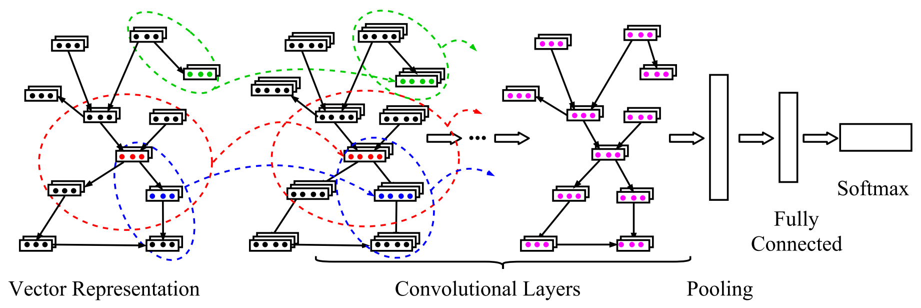


Figure 5.3: The architecture of the multi-layer convolutional neural network on graphs. Several steps of a convolution process is illustrated in the two first layers. The same color (red, green, or blue) of a node and an ellipse indicates the current position and the range of the filter.

several merge operations such as concatenation, element-wise multiplication, and element-wise maximization.

5.3 Directed Graph Convolutional Neural Networks

This section describes a multi-view multi-channel convolutional neural network (DGCNN) for labeled directed graph classification. Firstly, we formulate the graph classification problem.

A labeled directed graph is defined as $G = (V, E, \alpha)$ where V is the set of vertices, $E \subseteq V \times V$ is the set of directed edges, α is the vertex labeling function $\alpha : V \to \Sigma_V$ where Σ_V is the content of vertex labels. Given a set of training examples $T = \{(x_i, y_i)\}_{i=0}^{L}$ where $x_i \in \mathcal{X}$ is a graph, and $y_i \in \mathcal{Y} = \{+1, -1\}$ is a target label, the graph classification problem is to induce the mapping $f : \mathcal{X} \to \mathcal{Y}$

5.3.1 Convolutional Neural Networks on Directed Graphs

DGCNN is a general neural network architecture designed to treat directed graphs with vertex labels containing complex information. For example, in the CFG, each vertex is an instruction which may involve the instruction name, and several operands. Moreover, each instruction can be viewed in not only its contents but also other perspectives including instruction types or functions. Each information type of a vertex label is called a view. To leverage all available information corresponding to such characteristics of the graphs, the multi-view multi-layer convolutional neural network on directed graphs is developed.

Fig. 5.3 demonstrates the overview architecture of DGCNN. The first layer is used to generate vector representations (also called embeddings) for graph vertices, where each view of a vertex label is mapped into a real-valued vector in a n_f-dimensional space. Next several convolutional layers are stacked on the embedding layer to extract the features from different parts of the graph. We thereafter apply a dynamic pooling layer to gather extracted features over the entire graph before feeding to a fully-connected layer. Finally, an output layer is added to compute the categorical distributions for possible outcomes. For multi-class classification problems, softmax is selected as the activation function to convert final scores to probabilities of observing labels. In the remainder of this section, we explain in details about major layers of DGCNN including vector representations, convolutional and pooling layers.

5.3.2 Convolutional Layers

In a convolutional layer, we apply a set of circular filters with radius R sliding over graph structures to extract features for all locations on the graphs. Because each vertex contains several views of its label, the filters extend through the full views of the input volumes. For example, given a filter with $R = 2$, and a graph with 3 views, at each position, the filter is designed such that it covers a subgraph containing the current vertex and the neighbors, and extends to depth 3. In other words, each neuron in the convolutional layer is connected to a local region (subgraph) of the input and the connectivity is extended along edges and views.

Formally, during the forward pass, each filter slides through all vertices of the graph and computes dot products between entries of the filter and the input. Suppose that the subgraph in the sliding window includes $d + 1$ vertices (the current vertex and its neighbors) with vector representations of $x_0, x_1, ..., x_d \in \mathbb{R}^{v_f \times n_f}$, then the output of the filters is computed as follows:

$$y = tanh(\sum_{i=0}^{d} \sum_{j=1}^{v_f} W_{conv,i,j} \cdot x_{i,j} + b_{conv}) \tag{5.2}$$

where $y, b_{conv} \in \mathbb{R}^{v_c \times n_c}, W_{conv,i} \in \mathbb{R}^{v_c \times n_c \times v_f \times n_f}$. $tanh$ is the activation function. n_f and v_f are the vector size and the number of views of the input layer. n_c and v_c are the numbers of filters and views of the convolutional layer.

The problem is that because of arbitrary structures of graphs, the numbers of vertices in subgraphs are different. As can be seen in Fig.5.3, the current receptive field at the red node includes 5 vertices while only 3 vertices are considered if the window moves right down. Consequently, determining the number of weight matrices for filters is unfeasible. To deal with this obstacle, we divide vertices into groups and treat items in each group in a similar way. Regarding the way, the parameters for convolution have only three weight matrices including W^{cur}, W^{in}, and W^{out} for current, outgoing, and incoming nodes, respectively.

In the model, we stack several convolutional layers to broaden the area for extracting features of input graphs. Convolution preserves the input structures by using filters sliding over the entire graph. For this reason, the design procedure for all convolutional layers is the same. In the experiments, the networks have two convolutional layers with one or two views in the first convolution and one view in the second convolution. The filter sizes are set to 2. This means that at each position, considered objects involve the current vertex and its neighbors. To sum up, the set of parameters for DGCNN is

$\theta = \{[W^{cur}_{conv1}, W^{in}_{conv1}, W^{out}_{conv1}]_{v_i}, [W^{cur}_{conv2}, W^{in}_{conv2}, W^{out}_{conv2}], W_{hid}, W_{output}, [b_{conv1}]_{v_i}, b_{conv2}, b_{hid}, b_{output}\}$, where v_i is the number of views of the input data.

5.3.3 Dynamic Pooling

Convolutions preserve the spatial relationship between vertices by learning graph features using circular filters. After convolutions, the structure of the output is completely the same as that of the original one. Thus, the extracted features can not be fed directly to the fully-connected layer because of enormous and varying numbers among different graphs. An efficient solution to this problems is applying dynamic pooling [82] to normalize the features such that they have the same dimension.

In the model, we use one-way max pooling to gather the information from all parts of the graph to one fixed size vector regardless of graph shapes and sizes. The vector dimension is the number of filters in the last convolutional layer. Basically, in convolutional neural networks, pooling layers are applied to reduce the dimensionality of each feature map (the output of one filter) but retain the most important information. Pooling layers operate independently on every dimension of its input and resize the input spatially using an operation. Some types of operations are max, average, and sum. In the case of max pooling, the maximum value in each dimension is selected from the features. Instead of taking the largest element we could also take the average (average pooling) or the sum of all elements (sum pooling) in that window. In practice, max pooling has been shown to work better [57, 77, 96]. Therefore, the max pooling is adopted in DGCNN.

5.3.4 Training

We use mini-batch gradient descent algorithm for training the network. The objective is to minimize the mean square error loss function as follows:

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \tag{5.3}$$

where θ is the model parameters (Subsection 5.3.2), y_i is the label of data sample i, and $\hat{y}_i$ is the output of the network.

The training procedure is shown in the pseudo-code in algorithm 2. Firstly, the model parameters θ are randomly initialized. The training process is performed through a pre-defined number of epochs. For each loop, we calculate the loss function $J^{(i)}$ for each data sample $x^{(i)}$ separately according to Eq. 5.3. Then, back propagation algorithm is applied to compute the partial derivatives and evaluate the gradient. The model parameters are updated every mini-batch of n training examples.

5.3.5 Computational Complexity and Required Memory

By applying filters with flexible design, and dynamic pooling, DGCNN does not require any preprocessing such as padding to ensure graphs with the same number of nodes, and

Algorithm 2: Mini-batch gradient descent algorithm

```
Input  : Data samples $x^{(i)}$, $i = 1..N$;
         Learning rate $\eta$;
         Batch size $n$;
Output: Model parameters $\theta = \{W, B\}$
1  Randomly initialize $\theta$, $\Delta\theta \leftarrow 0$;
2  for $l \leftarrow 1$ to $nb\_epochs$ do
3      for $i \leftarrow 1$ to $N$ do
4          compute loss $J^{(i)}$;
5          compute the partial derivative $\frac{\partial J^{(i)}}{\partial \theta}$;
6          $\Delta\theta \leftarrow \Delta\theta$ + evaluate gradient($\frac{\partial J^{(i)}}{\partial \theta}$);
7          if $i\%\ n = 0$ or $i = N$ then
8              $\theta \leftarrow \theta - \eta\Delta\theta$;
9              $\Delta\theta \leftarrow 0$;
10         end
11     end
12 end
```

alignment to match corresponding nodes between graphs. Indeed, from the two first layers of the model (Fig. 5.3), filters slide over the entire graph and extract subgraph features at each location independently regardless of graph structures. Additionally, computing the feature map does not require any order of nodes (Eq. 5.2). Thus, the model can treat dynamic graphs and matching nodes among graphs is unnecessary.

DGCNN uses a constant amount of memory to store model parameters, and its runtime grows proportionally with the number of vertices, and vertex degrees. According to Eq. 5.2, the cost for computing the feature map in a convolutional layer is $(d_{max}+1) \times v_f \times n$, where d_{max} is the maximum degree of graphs, v_f is the number of views, and n is the number of graph nodes. In the pooling layer, we need $O(n)$ comparisons to gather all extracted features into a fixed-size vector. For hidden and output layers, the numbers of operations are constant. Intuitively, c and v_f are constant, with assumption of $d_{max} \ll n$ in large-scale graphs, the runtime complexity of DGCNN is $O[c \times (d_{max}+1) \times v_f \times n] + O(n) + O(1) = O(n \times d)$.

5.4 Experiments

To verify the performance of DGCNN in terms of accuracy and the ability to process large-scale graphs, we conduct experiments on two tasks including software defect prediction and malware analysis. To solve these problems, each data sample is converted into a directed graph of control flow (CFG) [3], and then DGCNN is utilized to build predictive models.

For software defect prediction, we formulate an end-to-end model with high accuracy

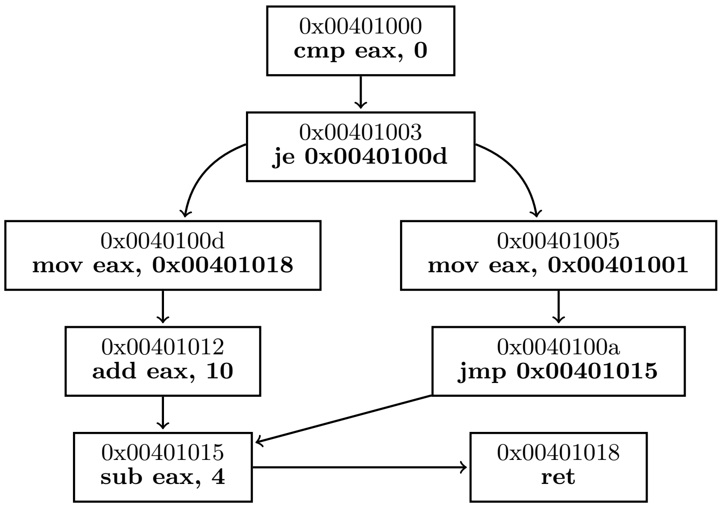


Figure 5.4: The Control Flow Graph of an assembly code fragment.

to predict the existence of defects in programming source code. For malware analysis, DGCNN shows the ability to handle CFGs with hundred thousands of nodes and edges.

We ran experiments on two systems including one node of a Fujitsu CX250 Cluster and one node of a SGI UV3000. For the Fujitsu CX250 Cluster, each node has two Intel Xeon processors E5-2680v2 2.80 GHz with ten cores, 64GB of RAM. For the SGI UV3000, each node has two Intel Xeon processors E5-4655v3 2.9GHz with six cores, 256GB of RAM.

5.4.1 Control Flow Graphs

A control flow graph (CFG) is a labeled directed graph, G = (V, E, α), where V is the set of vertices, $E \subseteq V \times V$ is the set of directed edges, α is the vertex labeling function $\alpha : V \to \Sigma_V$ where Σ_V is the contents of vertex labels. In CFGs, each $v \in V$ represents a basic block that is a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed); and $(v_i, v_j) \in E$ shows the control flow path from block v_i to block v_j (Fig. 5.4).

CFG analysis has been widely used for various problems because of showing execution sequences of programs. Compilers perform program optimization by determining the flow relationships based on within graph analysis [3]. Many studies have focused on mining CFGs to tackle the difficulties in malware analysis [5,12], software plagiarism [14,86]. For the two tasks in this study, software defects are hidden deeply in source code and only revealed while programs are running; malware uses different obfuscation techniques to prevent detection by anti-virus applications. Since CFGs show the behavior of programs, learning on CFGs may be beneficial for distinguishing patterns.

5.4.2 Vector representations

As mentioned before, an assembly instruction is viewed by two perspectives and it may have several operands. To take advantages of all information, the vector representation of each instruction is computed based on those of its components. Firstly, operands of block names, processor register names, and literal values are substituted by the symbols "name", "reg", and "val", respectively. Corresponding to the replacement, the instruction `addq $32, %rsp` has the form of `addq, value, reg`. After that, the vector of the instruction is determined as follows:

$$x = \begin{cases} x_{instruction\ token} \text{ (NoOp - without the use of operands)} \\ \frac{1}{C}\sum_{j=1}^{C} x_j \text{ (Op - with the use of operands),} \end{cases} \tag{5.4}$$

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

5.4.3 Software defect prediction

Task description

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. Deploying software products containing defects may cause serious consequences such as loss of money, time, and business credibility. 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 bug detection and localization is an urgent requirement in software industry. This helps to reduce the development efforts, and enhance the quality and reliability of software products.

In this study, we formulate a new approach to predict the existence of defects in source codes written in a programming language. Our proposed approach involves two steps: 1) generating the CFG representation, and 2) building classifiers. In the first step, each source file is compiled into an assembly code using g++ on Linux. The CFG thereafter is constructed to describe the execution flows of the assembly instructions. The second step leverages DGCNN to automatically learn defect features on CFG data.

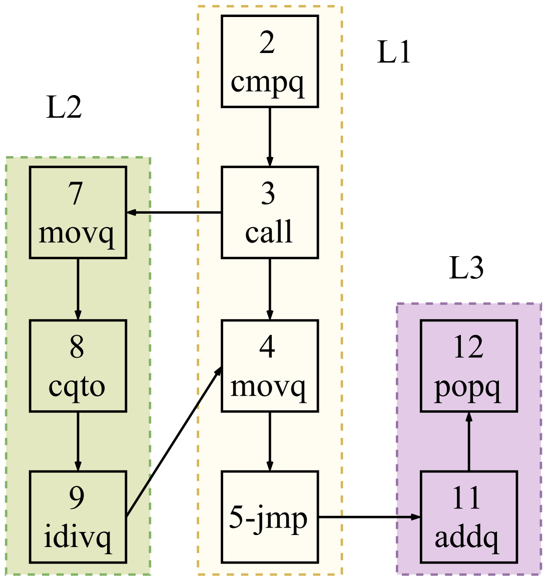
CFG construction

The CFG of a program is generated from its assembly code after compiling. Fig. 5.5 illustrates an example of the control flow graph constructed from an assembly code snippet, in which each vertex corresponds to an instruction and a directed edge shows the execution path from an instruction to the other.

The pseudo-code to generate CFGs is shown in Algorithm 3. The algorithm takes an assembly file as the input, and outputs the CFG. Building the CFG from an assembly code includes two major steps. In the first step, the code is partitioned into blocks of instructions based on the labels (e.g. `L1, L2, L3` in Fig. 5.5). The second step is creating the edges to represent the control flow transfers in the program. Specifically, the first line invokes procedure `initialize_Blocks` to read the file contents and return all the instruction blocks. In line 2, the set of edges is initially set to empty. From line 3 to 24, the graph edges are created by traversing all instructions of each block and considering

```
 1  .L1:
 2       cmpq     $0, 24(%rbp)
 3       call     .L2
 4       movq     16(%rbp), %rax
 5       jmp .L3
 6  .L2:
 7       movq     16(%rbp), %rax
 8       cqto
 9       idivq    24(%rbp)
10  .L3:
11       addq     $32, %rsp
12       popq     %rbp
```

(a)



(b)

Figure 5.5: An example of constructing CFGs from assembly code.(5.5a) a fragment of assembly code;(5.5b) the CFG of the code fragment (each node is viewed by the line number and the name of the instruction).

possible execution paths from the current instruction to others. For a block, because the instructions are executed in sequence, every node has an outgoing edge to the next one (line 5-9). Additionally, we consider two types of instructions which may have several targets. For `jump` instructions, an edge is added from the current instruction to the first one of the target block. We use two edges to model function calls, in which one is from the current node to the first instruction of the function and the other is from the final instruction of the function to the next instruction of the current node (line 10-24). Finally, the graphs are formed from the instruction and edge sets (line 25-26).

Datasets

The datasets were collected 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.

[1] `https://www.codechef.com/problems/<problem-name>`

Algorithm 3: The algorithm for constructing Control Flow Graphs from assembly code

```
Input  : $asm\_file$ - A file of assembly code
Output: The graph representation of the code
$blocks \leftarrow$ initialize_Blocks($asm\_file$);
$edges \leftarrow \{\}$;
for $i \leftarrow 0$ to $|blocks|$ do
    for $j \leftarrow 0$ to $|blocks[i].instructions|$ do
        if $j > 0$ then
            $inst\_1 \leftarrow blocks[i].instructions[j-1]$;
            $inst\_2 \leftarrow blocks[i].instructions[j]$;
            $edges.add$(new_Edge($inst\_1, inst\_2$));
        end
        if $inst\_1.type$=“jump” or $inst\_1.type$=“call” then
            $label \leftarrow inst\_1.params[0]$;
            $to\_block \leftarrow$find_Block_by_Label($label$);
            if $to\_block \neq NULL$ then
                $inst\_2 \leftarrow to\_block.first\_instruction$;
                $edges.add$(new_Edge($inst\_1, inst\_2$));
                if $inst\_1.type$=“call” then
                    $inst\_2 \leftarrow to\_block.last\_instruction$;
                    $inst\_1 \leftarrow inst\_1.next$;
                    $edges.add$(new_Edge($inst\_2, inst\_1$));
                end
            end
        end
    end
end
$instructions \leftarrow$ get_All_Instructions($blocks$);
return $construct\_Graph(instructions, edges)$;
```

- 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 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 files which are empty code, and unable to compile. To conduct experiments, each dataset is randomly split into three folds for training, validation, and testing by ratio 3:1:1.

Table 5.1 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. In addition, programs' CFGs vary considerably in size with the number of nodes from hundreds to thousands.

Table 5.1: Statistics of CodeChef datasets. The values are shown in form of average± standard deviation.

Dataset	Class					Nodes	Max nodes	Degree	Max degree
	0	1	2	3	4				
FLOW016	3,472	4,165	231	2,368	412	117±55	1,246	3.39±0.80	11
MNMX	5,157	3,073	189	113	213	211±296	3,073	3.72±1.86	43
SUBINC	3,263	2,685	206	98	232	142±63	1,245	3.19±0.63	17
SUMTRIAN	9,132	6,948	419	2,701	1,987	236±104	2,905	3.75±1.36	56

Baselines

We compare our models including DGCNN and ASCNN with various approaches including a tree based convolutional neural network (TBCNN) - a state-of-the-art deep neural model for this problem [66], sibling-subtree convolutional neural networks (SibStCNN), recursive neural networks (RvNN) [84], k nearest neighbors (kNN) with tree edit distance (TED) and Levenshtein distance (LD) [72], and support vector machines (SVMs) with bag-of-words (BoW) features. The parameters for the models are described as follows.

The neural networks. The structures of the networks are shown in Table 5.2. The networks share some initial parameters: the learning rate is 0.1, the token vectors have a size of 30, the batch size is 10.

k-nearest neighbors (kNN). The number of neighbors k is selected from {3, 5, 7, 9}. We found that $k = 3$ commonly reaches the highest performance on the validation sets.

SVM-BoW. Two parameters C and γ of the SVM with RBF kernel are tuned by using grid search.

Table 5.2: Structures and numbers of hyperparameters of the neural networks. Each layer is presented in form of the name followed by the number of neurons. Emb is a embedding layer. Rv, Conv, Pool, and FC stand for recursive, convolutional, pooling, and fully-connected, respectively.

Network	Architecture	weights	biases
RvNN	Coding30-Emb30-Rv600-FC600-Soft5	1,104,600	1,235
TBCNN	Coding30-Emb30-Conv600-GPool-FC600-Soft5	1,140,600	1,235
SibStCNN	Coding30-Emb30-Conv600-GPool-FC600-Soft5	1,140,600	1,235
ASCNN-1V	Conv600-Pool2-Conv300-GPool-FC300-Soft5	487,500	1,205
ASCNN-2V	[Conv600-Pool2-Conv300-GPool]×2-Merge-FC300-Soft5	973,500	2,105
DGCNN-1V	Conv100-Conv600-FC600-Soft5	552,000	1,305
DGCNN-2V	Conv100-Conv600-FC600-Soft5	561,000	1,305

Results

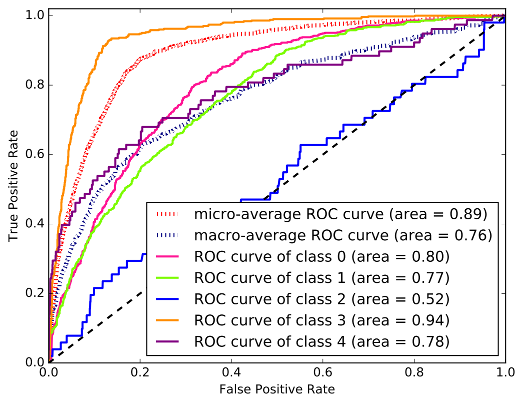
Table 5.3 shows the performance of classifiers according to the accuracy and F1 on the four datasets. Building the network models including the steps of training, validation and testing was within 24 hours. The best values of ASCNN and DGCNN are highlighted in bold, the best values of the other approaches are marked with superscipt $\star$ and in italic. As can be seen, assembly-based approaches significantly outperform others. Specifically, in comparison with the second best, DGCNN improves the accuracies and F1 scores by 12.39% and 12.18% on FLOW016, 1.2% and 1.28% on MNMX, 7.71% and 6.39% on SUBINC, and 1.98% and 0.5% on SUMTRIAN. Similarly, ASCNN achieves the higher accuracies and F1 scores by 10.37% and 10.94% on FLOW016, 1.48% and 1.63% on MNMX, 5.48% and 5.05% on SUBINC, and 2.1% and 2.21% on SUMTRIAN . 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 [93]. Therefore, it is impractical to manually design a set of good features which are able to distinguish faulty and non-faulty samples. Similarly, ASTs just represent the structures of source code. 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 instructions are equivalent to CPU-level instructions and CFGs are precise graphical structures which show behaviors of programs. As a result, applying DGCNN on CFGs and ASCNN on instruction sequences achieve the best accuracies and F1 scores on the experimental datasets about software defects.

Table 5.3: Comparison of classifiers according to accuracy and F1. 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.

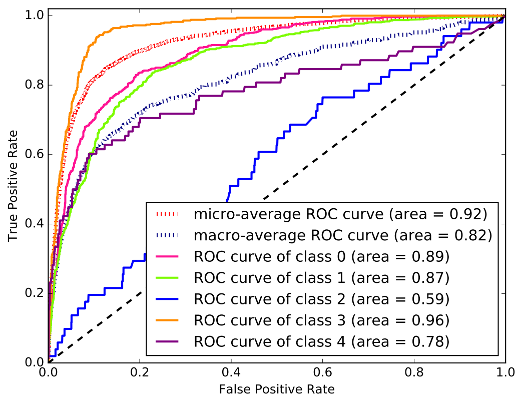
Approach	FLOW016		MNMX		SUBINC		SUMTRIAN	
	Acc.	F1	Acc.	F1	Acc.	F1	Acc.	F1
SVM-BoW	60.00	58.64	77.53	75.00	67.23	65.75	64.87	63.82
LD	60.75	60.61	79.13	77.89	66.62	66.36	65.81	65.73
TED	61.69	61.56	80.73	79.55	*68.31**	*68.03**	*66.97**	*66.83**
RvNN	61.03	58.98	82.56	80.48	64.53	62.07	58.82	56.29
TBCNN	*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_1V_NoOp	72.11	71.36	83.53	82.18	**73.79**	**73.08**	**69.07**	**69.04**
ASCNN_2V_NoOp	**73.47**	**72.79**	82.96	81.67	73.71	73.04	67.82	67.81
ASCNN_1V_Op	72.39	71.57	**84.33**	**82.67**	72.01	71.54	66.52	67.12
ASCNN_2V_Op	73.19	72.71	82.73	81.14	73.63	72.88	68.29	68.19
GCNN_1V_NoOp	73.80	72.57	83.19	81.28	70.93	69.61	68.83	**67.33**
GCNN_2V_NoOp	74.32	73.11	83.82	**82.32**	74.02	72.54	68.12	66.42
GCNN_1V_Op	**75.49**	**74.03**	**84.05**	82.28	72.40	70.67	68.19	65.91
GCNN_2V_Op	75.12	73.60	83.70	81.88	**76.02**	**74.42**	**68.95**	66.62

From the last eight rows of Table 5.3, the more the information is provided, the more efficient the learner is. In general, viewing an instruction by two perspectives including its contents and group may help boost DGCNN and ASCNN classifiers in both cases: with and without the use of operands. Similarly, taking into account of all components in instructions (Eq. 5.4) is beneficial. Specifically, DGCNN with one view reaches the accuracies of 75.49% on FLOW016, and 84.05% on MNMX; DGCNN with two views obtains the accuracies of 76.02% on SUBINC, and 68.95% on SUMTRIAN. ASCNN achieve better results on two datasets including FLOW016 and MNMX.

We also assess the effectiveness of the models in terms of the discrimination measure (AUC) which is equivalent to Wilcoxon test in ranking classifiers. For imbalanced datasets, many learning algorithms have a trend to bias the majority class due to the objective of error minimization. As a result, the models mostly predict an unseen sample as an instance of the majority classes, and ignore the minority classes. Fig. 5.6 plots the ROC curves of TBCNN and DGCNN_1V_NoOp classifiers on FLOW016 dataset, an imbalanced data with the minority classes of 2 and 4. Both two classifiers have a notable lower ability in detecting minority instances from the others. For predicting class 4, the TBCNN is even equivalent to a random classifier. After observing the other ROC curves we found the similar problem for all of the approaches on the experimental datasets. Thus, AUC is an essential measure for evaluating classification algorithms, especially in the case of imbalanced data.



(a)



(b)

Figure 5.6: The illustration of the discrimination ability between classes of classifiers on imbalanced datasets. Fig. 5.6a and Fig. 5.6b are the ROC curves of TBCNN and DGCNN_1V_NoOp on FLOW016 dataset, respectively.

Table 5.4: Performance comparison in terms of the AUC measure.

Approach	FLOW016	MNMX	SUBINC	SUMTRIAN
SVM-BoW	0.74	0.76	*0.73**	0.79
RvNN	0.75	*0.79**	0.69	0.73
TBCNN	*0.76**	0.77	0.72	0.78
SibStCNN	*0.76**	*0.79**	0.71	*0.80**
ASCNN_1V_NoOp	0.81	0.80	0.72	0.80
ASCNN_2V_NoOp	0.81	0.79	**0.75**	**0.81**
ASCNN_1V_Op	0.82	0.80	**0.75**	**0.81**
ASCNN_2V_Op	**0.83**	**0.82**	**0.75**	0.80
GCNN_1V_NoOp	**0.82**	**0.82**	0.74	**0.82**
GCNN_2V_NoOp	0.80	0.81	0.72	0.81
GCNN_1V_Op	0.81	0.80	**0.75**	0.81
GCNN_2V_Op	**0.82**	0.79	0.74	0.81

Table 5.4 presents the AUCs of probabilistic classifiers, which produce the probabilities or the scores to indicate the belonging degrees of an instance to classes. There are two main groups including assembly-based and tree-based approaches, in which the approaches in each group has the similar AUC scores; and assembly-based approaches show better performance than those of tree-based. It is worth noticing that, along with the efforts of accuracy maximization, the approaches based on assembly instructions also enhance the distinguishing ability between categories even on imbalanced data. Comparing with the second best, the DGCNN and ASCNN classifiers improve averages of 0.03 and 0.035 for the AUC measure.

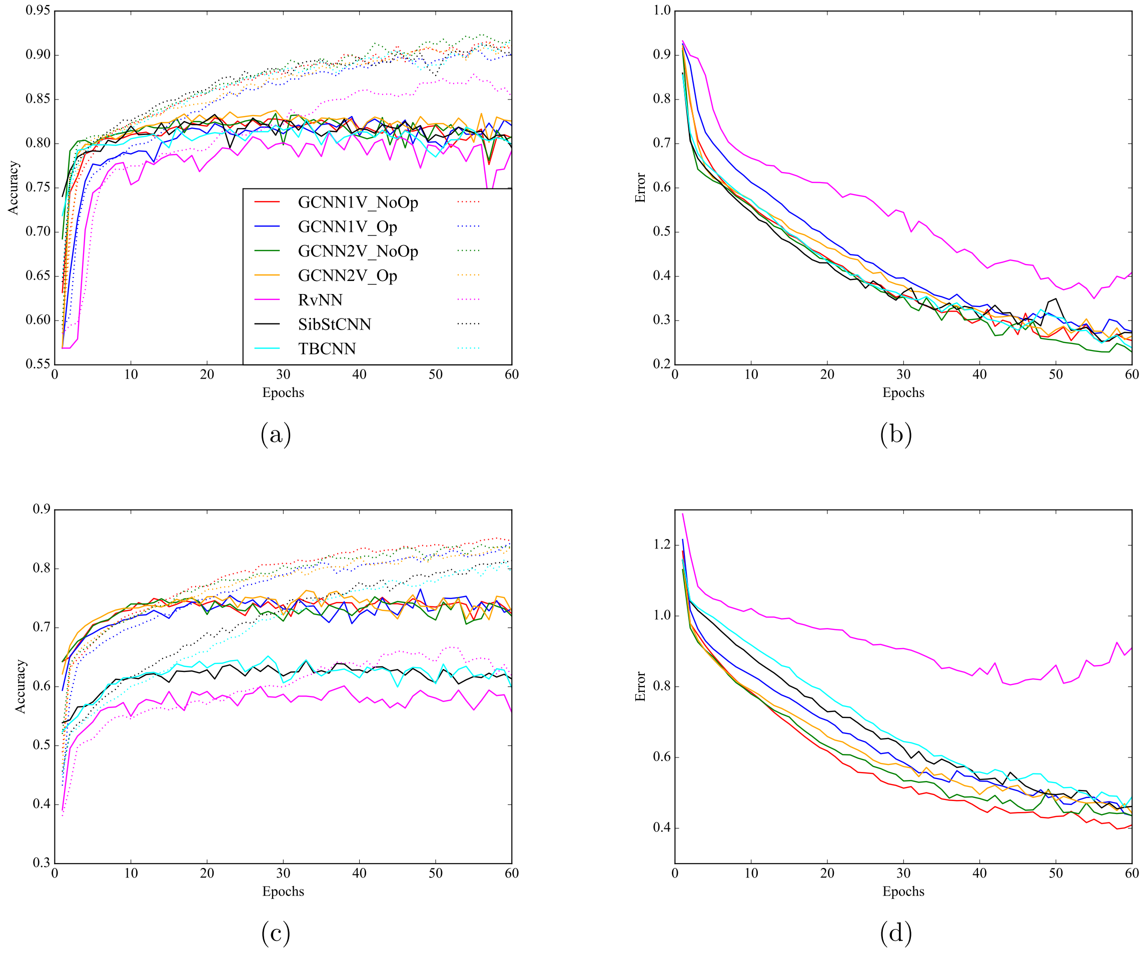


Figure 5.7: Learning curves of the networks. Fig. 5.7a and Fig. 5.7b, Fig. 5.7c and Fig. 5.7d are accuracy and error curves of MNMX and FLOW016 datasets; the solid and dot curves correspond to training and validation, respectively.

On the average, DGCNN usually shows the better performance than ASCNN and also significant outperforms the others. Fig. 5.7 plots the learning curves of the networks on two datasets MNMX and FLOW106. For all networks, the validation accuracies quickly converge to the optimal values after around 20 epochs and vary around such values in next epochs. We can see three groups of networks based on the learning curves. The first is the DGCNN's variants which reach highest accuracies for both training and validation, and have lowest errors. The second includes SibStCNN and TBCNN which their curves are closed to each other. The third is RvNN with the lowest accuracies and highest errors. From above analysis, we can conclude that leveraging precise control flow graphs of binary codes is suitable for software defect prediction, and DGCNN is a deep neural network for learning on labeled directed graphs efficiently.

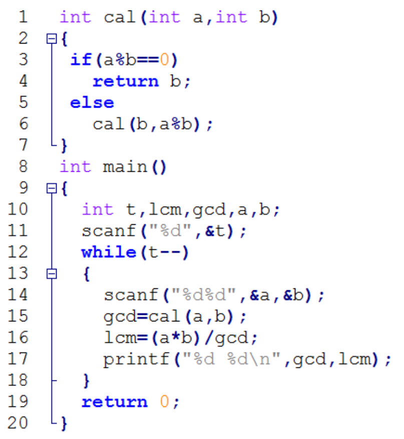
Error Analysis

We analyze cases of source code variations which methods are able to handle or not based on observations on classifiers' outputs, training and test data. We found that RvNN's performance is degraded when tree sizes increase. This problem is also pointed out from other research on tasks of natural language processing [83,84] and programming language processing [66]. From Tables 5.1, and 5.4 the larger trees, the lower accuracies and AUCs, RvNN obtains in comparison with other approaches, especially on SUMTRIAN dataset. SibStCNN and TBCNN obtain higher performance than other baselines due to learning features from subtrees. For analyzing tree-based methods in this section, we only take into account SibStCNN and TBCNN.

Effect of code structures: the tree-based approaches suffer from varying structures of ASTs. For example, given a program, we have many ways to reorganize the source code such as changing positions of some statements, constructing procedures and replacing statements by equivalent ones. These modifications lead to reordering the branches and producing new branches of ASTs (`File 3.c` and `File 4.c`). Because of the weight matrices for each node being determined based on the position, SibStCNN and TBCNN are easily affected by changes regarding tree shapes and sizes.

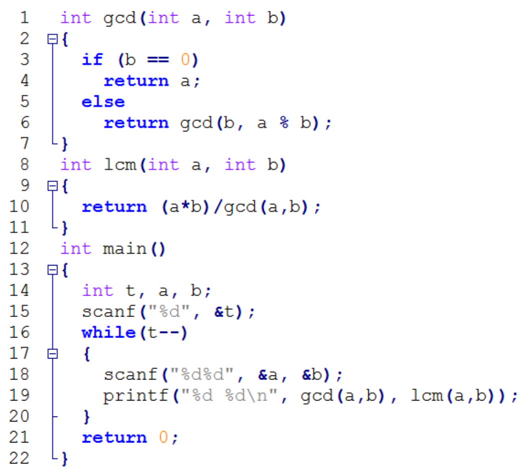
Meanwhile, graph-based approaches are able to handle these changes. We observed that although loop statements like `For`, `While`, and `DoWhile` have different tree representations, their assembly instructions are similar by using a jump instruction to control the loop. Similarly, moving a statement to possible positions may not result in notable changes in assembly code. Moreover, grouping a set of statements to form a procedure is also captured in CFGs by using edges to simulate the procedure invocation (Section 5.4.1).

Effect of changing statements: CFG-based approaches may be affected by replacements of statements. Considering source code in `File 3.c`, `File 5.c`, they have similar ASTs, but the assembly codes are different. In `C` language, statements are translated into different sets of assembly instructions. For example, with the same operator, the sets of instructions for manipulating data types of `int` and `long int` are dissimilar. Moreover, statements are possible replaced by others without any changes of program outcomes. Indeed, to show values, we can select either `printf` or `cout`. Since contents of CFG nodes are changed significantly, DGCNN may fail in predicting these types of variations.



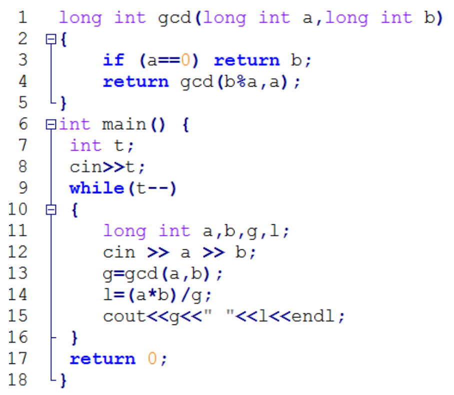
```
int cal(int a,int b)
{
 if(a%b==0)
   return b;
 else
   cal(b,a%b);
}
int main()
{
   int t,lcm,gcd,a,b;
   scanf("%d",&t);
   while(t--)
   {
     scanf("%d%d",&a,&b);
     gcd=cal(a,b);
     lcm=(a*b)/gcd;
     printf("%d %d\n",gcd,lcm);
   }
   return 0;
}
```

(a) File 3.c (a training sample)



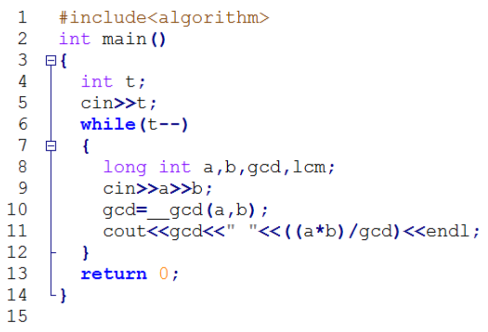
```
int gcd(int a, int b)
{
   if (b == 0)
     return a;
   else
     return gcd(b, a % b);
}
int lcm(int a, int b)
{
   return (a*b)/gcd(a,b);
}
int main()
{
   int t, a, b;
   scanf("%d", &t);
   while(t--)
   {
     scanf("%d%d", &a, &b);
     printf("%d %d\n", gcd(a,b), lcm(a,b));
   }
   return 0;
}
```

(b) File 4.c (**G+, T-**)



```
long int gcd(long int a,long int b)
{
     if (a==0) return b;
     return gcd(b%a,a);
}
int main() {
  int t;
  cin>>t;
  while(t--)
  {
     long int a,b,g,l;
     cin >> a >> b;
     g=gcd(a,b);
     l=(a*b)/g;
     cout<<g<<" "<<l<<endl;
  }
  return 0;
}
```

(c) File 5.c (**G-, T+**)



```
#include<algorithm>
int main()
{
   int t;
   cin>>t;
   while(t--)
   {
     long int a,b,gcd,lcm;
     cin>>a>>b;
     gcd=__gcd(a,b);
     cout<<gcd<<" "<<((a*b)/gcd)<<endl;
   }
   return 0;
}

```

(d) File 6.c (**G-, T-**)

Figure 5.8: Some source code examples in FLOW016 dataset which may cause mistakes of tree-based (T) and CFG-based (G) approaches. Fig. 5.8a is a sample in the training set. Figs. 5.8b, 5.8c, and 5.8d are samples in the test set. Symbols "+" and "-" denote the sample is correctly and incorrectly classified by the approaches.

Table 5.5: Statistics on malware dataset.

Dataset	Total	#Neg.	#Pos.	#nodes	Max nodes	Degree	Max degree
MALWARE	2,937	1,362	1,575	1,236±5,528	157,237	11.21±55.44	1736

Effect of using library procedures: when writing a source code, the programmer can use procedures from other libraries. In Fig. 5.8, `File 6.c` applies the procedure `__gcd` in the library `algorithm`, while the others use ordinary C statements for computing the greatest common divisor of each integer pair. Both ASTs and CFGs do not contain the contents of external procedures because they are not embedded to generate assembly code from source code. As a result, tree-based and graph-based approaches are not successful in capturing program semantics in these cases.

5.4.4 Malware analysis

Task description

To verify the ability to deal with large scale graphs, we apply DGCNN for malware analysis. The task is to check whether an executable file is malware. To solve this problem, a file is represented as a directed graph of control flow using a disassembler tool called BE-PUM (Binary Emulation for Pushdown Model) [34]. After that, several graph-based approaches are employed graph-based approaches to classify the data samples into malware and non-malware.

The dataset includes x86 binary files, wherein malware samples are supplied by LORIA, Loraine University [2], and VX Heaven [3]; and, non-malware files were collected from the system files in the folder *Windows*. From the last row of Table 5.5, the control flow graphs of files are very large with the number of nodes up to greater than 150,000. Because the data is quite small, the experiment is conducted using 5-fold cross validation.

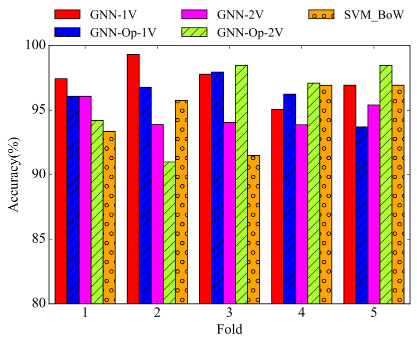
Baselines

For malware analysis, we compare DGCNN with SVM-Bow. We also investigated several graph kernel approaches such as Shortest Path Kernels and random walk graph kernels. However, these algorithms are unable to be executed due to computational complexity and required memory. According to previous studies, the kernel methods just process graphs with hundreds of nodes [11, 89].

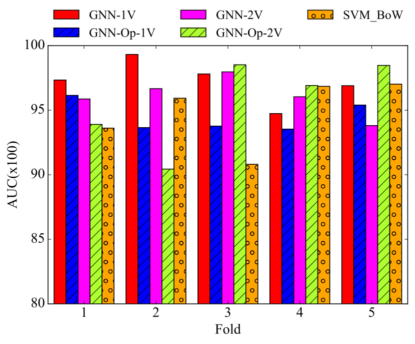
As mentioned in Subsection 5.3.5, DGCNN has $O(n \times d)$ time complexity, where n and d are the number of graph nodes and the max degree. In the case of malware dataset, the time complexity is equivalent to $O(n)$ because of $d \ll n$ (Table 5.5). Moreover, the required memory is about 3MB. As as results, both processes of training and testing DGCNN take only about 24 hours.

[2]`http://www.loria.fr/les-actus`

[3]`http://vxheaven.org/`



(a)



(b)

Figure 5.9: Comparison of DGCNN and SVM according to accuracy (5.9a) and AUC (5.9b) using 5-fold cross validation. The symbol *Op* indicates the use of operands.

Results

Fig. 5.9 depicts the performance of DGCNN in comparison with SVM-BoW in terms of accuracy and AUC. BE-PUM can generate precise control flow graphs of executable files under the presence of obfuscation techniques [34]. Thus using SVM with BoW features also obtains high results. One problem is the huge sizes of graphs that lead to obstacles in applying conventional methods for graphs. We tried SVM with shortest path kernels and random walk graph kernels, but running them was failed because of computational complexity and required memory. Meanwhile training the DGCNN models and predicting the testing sets take around 24 hours.

GCNN-1V achieves the best performance with the average accuracy of 97.31%, and the average AUC of 97.22%. By applying convolution to capture graphs' features, DGCNN is good at distinguishing between malware and non-malware files. Unlike the cases of software fault prediction, adding more information may degrade DGCNN's performance on the MALWARE dataset. This probably is caused by an increase of the number of tokens and the limited number of training instances. The numbers of unique tokens corresponding to the cases of with and without the use of operands are 1,427, 3,669, respectively, while the training data for each fold of cross validation contain about 2,350 samples.

5.5 Related Work

Graph data are known as one of complex structures and they have different types of representations. Various algorithms have been developed to tackle graph data. For labeled directed graphs, the existing approaches are based on subgraph isomorphism and graph kernels. However, these approaches not only are time and memory consuming but also require several constraints. Such problems lead to obstacles to adapting to practical applications. Gartner et al. proved that measuring graph similarity using subgraph isomorphism is NP-hard; the runtime grows exponentially with regard to the number of vertices. Although graph kernels are more efficient alternatives, measuring the similarities between graphs is computed in polynomial time. For graphs with n nodes and m edges, random walk kernels originally proposed by Gartner on labeled directed graphs have the running time of $O(n^6)$ [28]. Then, Vishwanathan et al. speeded up to $O(n^3)$ [89]; and, Kang et al. reduced the time complexity to $O(n^2)$ [44]. Several methods for computing graph kernels require the same number of graph nodes, and $O(m^2)$ memory. Because of above drawbacks, these algorithms are infeasible for graphs with more than *hundreds of nodes* [44].

Our work is diverse from several newly proposed convolutional neural networks on graphs such as graph-based convolutional neural networks for image classification [19, 23, 80], for text categorization [19], for classifying chemical compounds [67, 80]. These networks aim to solve problems that data samples are represented as weighted graphs. Applying these networks to several types of labeled graphs is impractical because the graphs must be converted into adjacency matrices. To obtain adjacency matrix representations, each node in a graph is assigned to a unique identifier and all graphs must share a set of identifiers. However, an instruction may appear at many locations in a CFG. This leads to the unknown of the common identifier set because it is impossible to align the vertices between CFGs. Indeed, given CFG_1 and CFG_2 with the sets of vertices $V_1 = \{cmp, mov, add\}$ and $V_2 = \{mov, mov, jmp\}$, we can not determine the node of the CFG_2 that corresponds the node *move* of the CFG_1.

Closely related are the works of Mou et al. [66] and Duvenaud et al. [22]. Mou et al. developed a tree-based convolutional neural network (TBCNN) to process abstract syntax trees (ASTs) of programming languages. ASTs can be viewed as a specific type of labeled directed graphs without cycles. Besides, TBCNN considers the position of each node to determine the corresponding weights in the convolution stage. Therefore, it is impossible to adapt TBCNN for graphs with cycles and arbitrary orders of nodes. Duvenaud et al. designed a graph-based architecture to encode invariant substructures in a molecule. Each molecule is represented as a graph with nodes being individual atoms and edges being bonds. To deal with dynamic structures of local regions in convolution stage, they combine the vectors the current node and neighbors by element-wise summation to form a single input vector. The weights of the input vector is determined according to the number of neighbors. Specifically, five sets of weights are used for local regions having 1-5 neighbors with the assumption that an atom has a maximum of 5 bonds. Extending this network to graphs in other domains faces some challenges. Unlike the graph of an atom, these graphs may have nodes with numerous neighbors. Thus, the network requires a huge number of weights causing overfitting. Moreover, embedding all node vectors into a single one ignores the relations in the substructures.

Our proposed network is a general framework for labeled graphs. To tackle dynamic substructures, we apply a shared parameter model in which the nodes having the same type share a set of weights and do not share any connections. We consider three types of nodes in a subgraph including the center, incoming and outgoing. In addition, our design can take multiple views of nodes to enrich the data for learning.

Conclusions

In this dissertation, I present some new models and techniques for source code analysis. Chapter 3 introduced a combination model for feature weighting and parameter optimization. In most of data sets, the relevance between features and class labels are different. Thus, quantifying the extent of the relevance will help us in building more accurate classifiers. Our proposed model is an appropriate tool for optimizing classifiers' parameters and features' weights because it can boost the performance notably and overcome the disadvantages of imbalanced data.

Chapter 4 turned to a more convenient approach: the applying of deep learning on abstract syntax trees (ASTs). An advantage of deep neural networks is that they do not require any handcrafted features. In machine learning, the accuracy of methods is greatly influenced by the quality of input data. Since containing rich information of programs, ASTs should be utilized to solve program analysis problems. We surveyed various machine learning algorithms from lazy learners like kNN with tree edit distance (TED) to deep neural networks like TBCNN, and found that AST-based methods completely beat the metrics-based. ASTs are high-dimensional data with the node numbers up to thousands. To refine the ASTs, we presented a technique to prune redundant branches and reconstruct subtrees. This leads to an increase in the classification accuracy and a decrease in running time. Additionally, several first layers of deep neural networks are to learn good features of the input data. After obtaining such features, we can feed to feed-forward layers or other learning algorithms to compute the distribution probabilities. Regarding this, we proposed some combinations of deep neural networks and common learning algorithms to make more powerful models.

Chapter 5 explored deeper semantic meanings of programs: the applying of deep learning on assembly code. ASTs just show the structures of programs. Thus, they may fail when adapting to other problems that need to discover the behavior of programs such as software defect prediction. Meanwhile, assembly code exposes the execution flow of a program. We designed two convolutional architectures to learn defect features from sequences and control flow graphs. Our approaches outperform the others that based on features and trees.

Bibliography

[1] Fumio Akiyama. An example of software system debugging. In *IFIP Congress (1)*, volume 71, pages 353–359, 1971.

[2] Hamoud Aljamaan, Mahmoud O Elish, and Irfan Ahmad. An ensemble of computational intelligence models for software maintenance effort prediction. In *Advances in Computational Intelligence*, pages 592–603. Springer, 2013.

[3] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5-7, pages 1–19. ACM, 1970.

[4] Marcos Alvares, Tshilidzi Marwala, and Fernando Buarque de Lima Neto. Application of computational intelligence for source code classification. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 895–902. IEEE, 2014.

[5] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. Graph-based malware detection using dynamic analysis. *Journal in computer virology*, 7(4):247–258, 2011.

[6] Gustavo EAPA Batista and Maria Carolina Monard. An analysis of four missing data treatment methods for supervised learning. *Applied Artificial Intelligence*, 17(5-6):519–533, 2003.

[7] Rey Mark John SA Bautista, Vishnu Joshua L Navata, Aldrich H Ng, Ma Santos, S Timothy, Justine D Albao, and Edison A Roxas. Recognition of handwritten alphanumeric characters using projection histogram and support vector machine. In *Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment and Management (HNICEM), 2015 International Conference on*, pages 1–6. IEEE, 2015.

[8] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377. IEEE, 1998.

[9] Philip Bille. A survey on tree edit distance and related problems. *Theoretical computer science*, 337(1):217–239, 2005.

[10] David Binkley and Mark Harman. Results from a large-scale study of performance optimization techniques for source code analyses based on graph reachability algorithms. In *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*, pages 203–212. IEEE, 2003.

[11] Karsten M Borgwardt and Hans-Peter Kriegel. Shortest-path kernels on graphs. In *Data Mining, Fifth IEEE International Conference on*, pages 8–pp. IEEE, 2005.

[12] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 129–143. Springer, 2006.

[13] Cagatay Catal. Software fault prediction: A literature review and current trends. *Expert systems with applications*, 38(4):4626–4636, 2011.

[14] Dong-Kyu Chae, Jiwoon Ha, Sang-Wook Kim, BooJoong Kang, and Eul Gyu Im. Software plagiarism detection: a graph-based approach. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, pages 1577–1580. ACM, 2013.

[15] E Chandra and P Edith Linda. Class break point determination using ck metrics thresholds. *Global journal of computer science and technology*, 10(14), 2010.

[16] Chih-Chung Chang and Chih-Jen Lin. Libsvm: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.

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

[18] Lawrence Davis. Handbook of genetic algorithms. 1991.

[19] 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 3837–3845, 2016.

[20] Erik D Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. An optimal decomposition algorithm for tree edit distance. *ACM Transactions on Algorithms (TALG)*, 6(1):2, 2009.

[21] Carlotta Domeniconi, Jing Peng, and Dimitrios Gunopulos. Locally adaptive metric nearest-neighbor classification. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(9):1281–1285, 2002.

[22] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on

graphs for learning molecular fingerprints. In *Advances in neural information processing systems*, pages 2224–2232, 2015.

[23] Michael Edwards and Xianghua Xie. Graph based convolutional neural network. *arXiv preprint arXiv:1609.08965*, 2016.

[24] Weiguo Fan, Edward A Fox, Praveen Pathak, and Harris Wu. The effects of fitness functions on genetic programming-based ranking discovery for web search. *Journal of the American Society for Information Science and Technology*, 55(7):628–636, 2004.

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

[26] Honghai Feng, Guoshun Chen, Cheng Yin, Bingru Yang, and Yumei Chen. A svm regression based approach to filling in missing values. In *Knowledge-Based Intelligent Information and Engineering Systems*, pages 581–587. Springer, 2005.

[27] Giles M Foody. The effect of mis-labeled training data on the accuracy of supervised image classification by svm. In *Geoscience and Remote Sensing Symposium (IGARSS), 2015 IEEE International*, pages 4987–4990. IEEE, 2015.

[28] Thomas Gärtner, Peter Flach, and Stefan Wrobel. On graph kernels: Hardness results and efficient alternatives. In *Learning Theory and Kernel Machines*, pages 129–143. Springer, 2003.

[29] Holland Goldberg, David E and John H. Genetic algorithms and machine learning. *Machine learning*, 3(2):95–99, 1988.

[30] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[31] 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 *EANN*, volume 2009, pages 223–234. Springer, 2009.

[32] Jerzy W Grzymala-Busse, Linda K Goodwin, Witold J Grzymala-Busse, and Xinqun Zheng. Handling missing attribute values in preterm birth data sets. In *Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing*, pages 342–351. Springer, 2005.

[33] Akkus Guvenir, H Altay and Aynur. Weighted k nearest neighbor classification on feature projections. In *Proceedings of the 12-th International Symposium on Computer and Information Sciences, Antalya, Turkey*, 1997.

[34] Nguyen Minh Hai, Mizuhito Ogawa, and Quan Thanh Tho. Obfuscation code localization based on cfg generation of malware. In *International Symposium on Foundations and Practice of Security*, pages 229–247. Springer, 2015.

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

[36] Sten Hjelmqvist. Fast, memory efficient levenshtein algorithm, 2014.

[37] Lin Chih-Jen Hsu Chih-Wei, Chang Chih-Chung. A practical guide to support vector classification, 2003.

[38] Sun-Jen Huang, Nan-Hsing Chiu, and Li-Wei Chen. Integration of the grey relational analysis with genetic algorithm for software effort estimation. *European Journal of Operational Research*, 188(3):898–909, 2008.

[39] Liangxiao Jiang, Chaoqun Li, Shasha Wang, and Lungan Zhang. Deep feature weighting for naive bayes and its application to text classification. *Engineering Applications of Artificial Intelligence*, 52:26–39, 2016.

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

[41] James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282. ACM, 2005.

[42] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[43] Stephen H Kan. *Metrics and models in software quality engineering.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[44] U Kang, Hanghang Tong, and Jimeng Sun. Fast random walk graph kernel. In *Proceedings of the 2012 SIAM International Conference on Data Mining*, pages 828–838. SIAM, 2012.

[45] Jaswinder Kaur, Satwinder Singh, Karanjeet Singh Kahlon, and Pourush Bassi. Neural network-a novel technique for software effort estimation. *International Journal of Computer Theory and Engineering*, 2(1):17, 2010.

[46] Taghi M Khoshgoftaar and Kehan Gao. Feature selection with imbalanced data for software defect prediction. In *Machine Learning and Applications, 2009. ICMLA'09. International Conference on*, pages 235–240. IEEE, 2009.

[47] 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.

[48] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[49] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *International Static Analysis Symposium*, pages 40–56. Springer, 2001.

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

[51] Chang-Hwan Lee. A gradient approach for value weighted classification learning in naive bayes. *Knowledge-Based Systems*, 85:71–79, 2015.

[52] Pornchai Lerthathairat and Nakornthip Prompoon. An approach for source code classification to enhance maintainability. In *Computer Science and Software Engineering (JCSSE), 2011 Eighth International Joint Conference on*, pages 319–324. IEEE, 2011.

[53] 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.

[54] Bozhao Li, Na Chen, Jing Wen, Xuebo Jin, and Yan Shi. Text categorization system for stock prediction. *International Journal of u-and e-Service, Science and Technology*, 8(2):35–44, 2015.

[55] 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.

[56] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 872–881. ACM, 2006.

[57] Lingqiao Liu, Lei Wang, and Xinwang Liu. In defense of soft-assignment coding. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2486–2493. IEEE, 2011.

[58] Kenneth C Louden et al. *Programming languages: principles and practices*. Cengage Learning, 2011.

[59] Lowe and David G. Similarity metric learning for a variable-kernel classifier. *Neural computation*, 7(1):72–85, 1995.

[60] 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.

[61] Atif M Memon and Qing Xie. Studying the fault-detection effectiveness of gui test cases for rapidly evolving software. *IEEE transactions on software engineering*, 31(10):884–896, 2005.

[62] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *Software Engineering, IEEE Transactions on*, 33(1):2–13, 2007.

[63] Mitchell and Melanie. *An introduction to genetic algorithms*. MIT press, 1998.

[64] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2012.

[65] Alessandro Moschitti. Efficient convolution kernels for dependency and constituent syntactic trees. In *ECML*, volume 4212, pages 318–329. Springer, 2006.

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

[67] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In *Proceedings of the 33rd annual international conference on machine learning. ACM*, 2016.

[68] Roberto Paredes and Enrique Vidal. A class-dependent weighted dissimilarity measure for nearest neighbor classification problems. *Pattern Recognition Letters*, 21(12):1027–1036, 2000.

[69] Mateusz Pawlik and Nikolaus Augsten. Rted: a robust algorithm for the tree edit distance. *Proceedings of the VLDB Endowment*, 5(4):334–345, 2011.

[70] Mateusz Pawlik and Nikolaus Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157–173, 2016.

[71] Viet Anh Phan and Lam Thu Bui. Genetic algorithm and application for supporting working schedule at hospitals. *LQDTU Journal of Science and Technology: The Section on Information and Communication Technology (LQDTU-JICT)*, 2:92–104, 4/2013.

[72] 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.

[73] William F Punch III, Erik D Goodman, Min Pei, Lai Chia-Shun, Paul D Hovland, and Richard J Enbody. Further research on feature selection and classification using genetic algorithms. In *ICGA*, pages 557–564, 1993.

[74] Michael L Raymer, William F Punch, Erik D Goodman, Leslie Kuhn, Anil K Jain, et al. Dimensionality reduction using genetic algorithms. *Evolutionary Computation, IEEE Transactions on*, 4(2):164–171, 2000.

[75] 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.

[76] José A Sáez, Joaquín Derrac, Julián Luengo, and Francisco Herrera. Statistical computation of feature weighting schemes through data estimation for nearest neighbor classifiers. *Pattern Recognition*, 47(12):3941–3948, 2014.

[77] Tara N Sainath, Ron J Weiss, Andrew W Senior, Kevin W Wilson, and Oriol Vinyals. Learning the speech front-end with raw waveform cldnns. In *INTERSPEECH*, pages 1–5, 2015.

[78] J. Sayyad Shirabad and T.J. Menzies. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005.

[79] Gordon Schulmeyer and James I McManus. *Handbook of software quality assurance.* Van Nostrand Reinhold Co., 1987.

[80] Martin Simonovsky and Nikos Komodakis. Dynamic edge-conditioned filters in convolutional neural networks on graphs. *arXiv preprint arXiv:1704.02901*, 2017.

[81] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[82] Richard Socher, Eric H Huang, Jeffrey Pennin, Christopher D Manning, and Andrew Y Ng. Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. In *Advances in Neural Information Processing Systems*, pages 801–809, 2011.

[83] 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.

[84] Richard Socher, Alex Perelygin, Jean Y Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, Christopher Potts, et al. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the conference on empirical methods in natural language processing (EMNLP)*, volume 1631, page 1642, 2013.

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

[86] Xin Sun, Yibing Zhongyang, Zhi Xin, Bing Mao, and Li Xie. Detecting code reuse in android applications using component-based control flow graph. In *IFIP International Information Security Conference*, pages 142–155. Springer, 2014.

[87] Duyu Tang, Bing Qin, and Ting Liu. Document modeling with gated recurrent neural network for sentiment classification. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1422–1432, 2015.

[88] Secil Ugurel, Robert Krovetz, and C Lee Giles. What's the code?: automatic classification of source code archives. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 632–638. ACM, 2002.

[89] S Vichy N Vishwanathan, Nicol N Schraudolph, Risi Kondor, and Karsten M Borgwardt. Graph kernels. *Journal of Machine Learning Research*, 11(Apr):1201–1242, 2010.

[90] Romi Satria Wahono and Nanna Suryana. Combining particle swarm optimization based feature selection and bagging technique for software defect prediction. *International Journal of Software Engineering and Its Applications*, 7(5):153–166, 2013.

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

[92] 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.

[93] 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.

[94] Jia Wu, Shirui Pan, Xingquan Zhu, Zhihua Cai, Peng Zhang, and Chengqi Zhang. Self-adaptive attribute weighting for naive bayes classification. *Expert Systems with Applications*, 42(3):1487–1502, 2015.

[95] Zhong-Liang Xiang, Xiang-Ru Yu, and Dae-Ki Kang. Experimental analysis of naïve bayes classifier based on an attribute weighting framework with smooth kernel density estimations. *Applied Intelligence*, pages 1–10, 2015.

[96] Jianchao Yang, Kai Yu, Yihong Gong, and Thomas Huang. Linear spatial pyramid matching using sparse coding for image classification. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 1794–1801. IEEE, 2009.

[97] Jun Zheng. Cost-sensitive boosting neural networks for software defect prediction. *Expert Systems with Applications*, 37(6):4537–4543, 2010.

Publications

JOURNALS

[1] Anh Viet Phan, Minh Le Nguyen, Lam Thu Bui, **Feature weighting and SVM parameters optimization based on genetic algorithms for classification problems** , Applied Intelligence, 2016

[2] Anh Viet Phan, Phuong Ngoc Chau, Minh Le Nguyen, Lam Thu Bui, **Automatically classifying source code using tree-based approaches**, Data & Knowledge Engineering, 2017

[3] Anh Viet Phan, Minh Le Nguyen, Yen Lam Hoang Nguyen, Lam Thu Bui, **DGCNN: A Convolutional Neural Network over Large-scale Labeled Graphs**, Journal of Engineering Applications of Artificial Intelligence, 2017 (Submitted)

INTERNATIONAL CONFERENCES

[1] Viet Anh Phan, Ngoc Phuong Chau, Minh Le Nguyen, **Exploiting Tree Structures for Classifying Programs by Functionalities**, The Eighth International Conference on Knowledge and Systems Engineering (KSE 2016)

[2] Ngoc Phuong Chau, Viet Anh Phan, Minh Le Nguyen, **Deep Learning and Sub-Tree Mining for Document Level Sentiment Classification**, The Eighth International Conference on Knowledge and Systems Engineering (KSE 2016)

[3] Minh-Tien Nguyen, Viet-Anh Phan, Truong-Son Nguyen, Minh-Le Nguyen, **Learning to Rank Questions for Community Question Answering with Ranking SVM**, ECML/PKDD 2016 Discovery Challenge - cQA Challenge: Learning to re-rank questions for community question answering.

[4] Anh Viet Phan, Minh Le Nguyen, Lam Thu Bui, **SibStCNN and TBCNN + kNN-TED: New Models over Tree Structures for Source Code Classification**, International Conference on Intelligent Data Engineering and Automated Learning (IDEAL'2017)

[5] Anh Viet Phan, Minh Le Nguyen, Lam Thu Bui, **Convolutional Neural Networks over Control Flow Graphs for Software Defect Prediction**, International Conference on Tools with Artificial Intelligence (ICTAI2017)

[6] Anh Viet Phan, Minh Le Nguyen, Lam Thu Bui, **Convolutional Neural Networks on Assembly Code for Predicting Software Defects**, Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES 2017)

AWARDS

[1] Best Student Paper, received in The Eighth International Conference on Knowledge and Systems Engineering, 06-08 October, 2016 – Ha Noi, Viet Nam

[2] Second-best System of the cQA Challenge: Learning to re-rank questions for community question answering, received in the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery (ECMLPKDD), 19-23 September, 2016, Riva Del Garda, Italy