Data & Knowledge Engineering xxx (xxxx) xxx–xxx

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

Automatically classifying source code using tree-based approaches

Anh Viet Phan[a,b], Phuong Ngoc Chau[a], Minh Le Nguyen[a,*], Lam Thu Bui[b]

[a] *Japan Advanced Institute of Information Technology, Ishikawa, Japan*
[b] *Le Quy Don Technical University, 236 Hoang Quoc Viet Rd., Ha Noi, Viet Nam*

A R T I C L E I N F O

Keywords:
Abtract Syntax Tree (AST)
Tree-based convolutional neural networks(TBCNN)
Support Vector Machines (SVMs)
K-Nearest Neighbors (kNN)

A B S T R A C T

Analyzing source code to solve software engineering problems such as fault prediction, cost, and effort estimation always receives attention of researchers as well as companies. The traditional approaches are based on machine learning, and software metrics obtained by computing standard measures of software projects. However, these methods have faced many challenges due to limitations of using software metrics which were not enough to capture the complexity of programs.

To overcome the limitations, this paper aims to solve software engineering problems by exploring information of programs' abstract syntax trees (ASTs) instead of software metrics. We propose two combination models between a tree-based convolutional neural network (TBCNN) and k-Nearest Neighbors (kNN), support vector machines (SVMs) to exploit both structural and semantic ASTs' information. In addition, to deal with high-dimensional data of ASTs, we present several pruning tree techniques which not only reduce the complexity of data but also enhance the performance of classifiers in terms of computational time and accuracy.

We survey many machine learning algorithms on different types of program representations including software metrics, sequences, and tree structures. The approaches are evaluated based on classifying 52000 programs written in C language into 104 target labels. The experiments show that the tree-based classifiers dramatically achieve high performance in comparison with those of metrics-based or sequences-based; and two proposed models TBCNN + SVM and TBCNN + kNN rank as the top and the second classifiers. Pruning redundant AST branches leads to not only a substantial reduction in execution time but also an increase in accuracy.

1. Introduction

Program classification as well as other software engineering problems are hot research topics, receiving the attention of researchers and companies due to a wide range of applications. In practice, software components in a project are usually categorized into repositories according to several particular criteria. Properly organizing software repositories is beneficial to programmer cooperation, maintenance and reuse of software. These are important factors which enhance software quality, reduce development time and efforts. However, manually assigning a software component into the repository compatible with the criteria is impractical because sizes of projects are very large and increase rapidly [1]. Thus, in the software industry, building a tool to automatically categorize source code repositories is an urgent requirement to manage big projects. Increasingly more researchers are showing interest in applying machine learning techniques to tackle this problem. However, how to extract suitable features from programs in order to improve the learning performance has been a challenging question.

Most studies have focused on using software metrics to build classifiers [1–3]. The metrics-based approaches are also widely

* Corresponding author.

http://dx.doi.org/10.1016/j.datak.2017.07.003
Received 20 January 2017; Received in revised form 11 July 2017; Accepted 17 July 2017
0169-023X/ © 2017 Published by Elsevier B.V.

applied to solve various software engineering problems such as fault prediction, cost, and effort estimation [4–7]. 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 [8]; and most of the currently used metrics have multiple definitions and ambiguous counting rules [9]. Akiyama et al. [10] predicted defects from lines of code (LOC) by Eq. (1).

$$\#of\ \ defects = 4.86 + 0.018*LOC \tag{1}$$

Halstead et al. [11] proposed several complexity metrics and used these as predictors of program defects. The most notable predictor asserted by the author is computed based on number of unique operators and unique operands as follows.

$$\#of\ \ defects = volume/3000 \tag{2}$$

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 [12] is around 71% [8,13].

Recently, several researchers proposed applying language processing approaches to address software engineering problems [14–17]. These new methods have shown high effectiveness in different tasks such as software fault prediction, program classification and summarization of source code. Binkley et al. studied the Information Retrieval (IR) based method for predicting faults in modules [17]. Huo et al. employed a convolutional neural network (CNN), which can capture the semantics of the program from both lexical and program structures for automatically locating the potential buggy source code [15]. Mou et al. proposed a novel TBCNN, which uses subtree detectors to extract structural information of tree structures. His model was successfully in processing ASTs of programs to deal with two program analysis tasks: classifying programs by functionalities, and detecting source code snippets of specific patterns [16].

In this paper, due to containing full and extra information of programs, ASTs are selected as the input data for machine learning techniques instead of using software metrics. The advantages of the use of ASTs are detailed in Section 3.2. We address the program classification problem by employing various techniques to explore information of programs' ASTs such as tree-based convolution kernels, tree edit distance, tree kernels, and Levenshtein distance. Machine learning techniques then are applied to build classifiers based on the extracted information. Furthermore, we propose two combination models including TBCNN + kNN-TED and TBCNN + SVM to extract both structural and underlying information of ASTs to enhance the accuracy of classifiers. In the TBCNN + kNN-TED model, the label of an unseen instance is decided based on both the output probabilities of TBCNN and the normalized distances to its neighbors. Unlikely, in the TBCNN + SVM model, TBCNN is used to automatically generate features from source code and then these features are classified by SVM classifiers. The performance of this approach was validated by classifying programs according to functionalities. Similarly, these methods can also be applied to other software engineering problems if the input data is in the same format (source code of programming languages with grammars).

When processing ASTs structures, we usually face a problem of high-dimensional data. The size of an AST increases more than linearly when the program size is larger. In the experimental dataset, the largest AST contains 7027 nodes, while the program has only 343 lines of code including comments and blank lines. High-dimensional data not only lead to waste of time and memory but also affect the performance of algorithms. Thus, dimension reduction is an essential task for the AST-based approach. For this reason, we propose heuristic techniques to prune redundant branches and reconstruct sub-trees of ASTs. The experiments show that such technique results in a decrease of execution time and an increase in classification accuracy as well. It is interesting that, due to the refinement of the input data, we can detect 356 source code clones by using tree edit distance (TED) to measure the similarity between ASTs. It means that many programs were completely copied or copied with tiny modifications from the others. We are going to describe more details about this finding in Section 6.

The main contribution of this paper can be summarized as follows:

1. Surveying and comparing many machines algorithms which work on various types of input data such as software metrics, sequences, and tree structures to a software engineering problem.
2. Proposing several techniques to refine input data by pruning redundant branches and reconstructing sub-trees of ASTs;
3. Proposing two combination models of TBCNN and kNN, SVM to enhance classification accuracy.

The remainder of the paper is organized as follows: Section 2 presents some relevant studies on source code classification in specifically and software engineering problems in generally. Section 3 briefly introduces the definition of abstract syntax trees (ASTs) and several machine learning algorithms for classifying source code. Section 4 describes two proposed models to enhance classification accuracy. The data, data preprocessing techniques and the experimental setup of case studies are described in Section 5. Section 7 analyzes some examples of source code which motivate us to apply pruning and reconstructing ASTs. We evaluate the results of the methods in Section 6 and conclude in Section 8.

2. Related work

Source code analysis has been widely applied to a variety of software engineering tasks such as clone detection [18,19], fault location [20,21], quality assessment [22,23], 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. [1] 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. [24] 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. [25] proposed an approach that classifies source code with software metrics and fuzzy logic and then improves bad smell, ambiguous code. Chandra et al. [26] 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 [27]. 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. [28] presented a collection of techniques which employ graphs as internal representations to improve source code analysis tools. Komondoor et al. [29] 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. [30] 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. [31] 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. [16] 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.

In this paper, 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.

3. Preliminaries

This section introduces a problem of classifying programs by functionalities and several approaches for solving this problem.

3.1. Program classification

Program classification is grouping programs according to specific criteria. Developing tools for these problems has many important applications in the field of software engineering. For example, program classification by functionalities is beneficial to managing big projects because new components are automatically tagged into the suitable repository. Similarly, categorizing a large amount of open source code on web, e.g. GitHub (https://github.com/), by application topics and programming languages is beneficial to software reuse.

In this work, we apply some approaches for classifying source code written in C++ language, in which programs with the same label perform the similar tasks.

3.2. Abstract syntax tree

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. 1 illustrates the AST of the `C` statement "`printf`("`The sum of` x + y =% d", x + y);".

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, semicolons, 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.

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

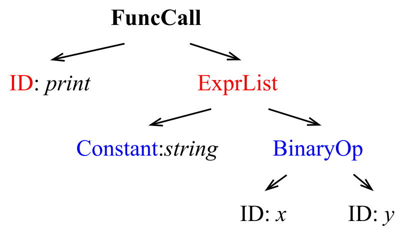

Fig. 1. The AST of the statement "`printf`("`The sum of` x + y =% d", x + y);".

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.

3.3. Tree edit distance

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 [32]. 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'.

Fig. 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 [33–35]. In this paper, we apply a robust and memory-efficient algorithm for the tree edit distance namely AP-TED (All Path Tree Edit Distance) [35] to compute the distance between AST trees.

3.4. Tree-based convolutional neural network

Tree-based convolutional neural network (TBCNN) is a novel model proposed by Mou [16], which showed a notable performance on program classification problem. Fig. 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 $x \in \Re_{N_f}$. Then the author designed a set of fixed-depth subtree detectors sliding over entire AST to extract structural information of the program. The output of the feature detectors is computed by the following equation.

$$y = \tanh\left(\sum_{i=1}^{n} W_{conv,i} \cdot x_i + b_{conv}\right) \tag{3}$$

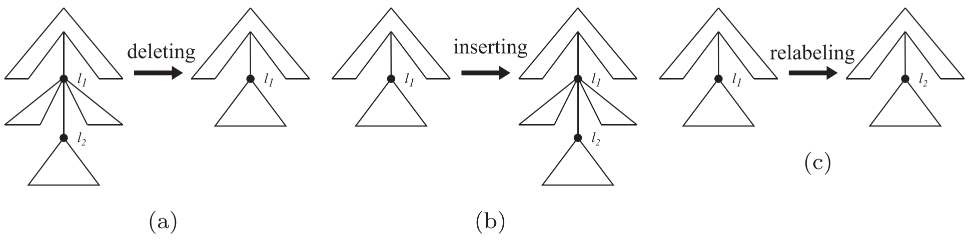

Fig. 2. Basic tree edit operations. (2a) Deleting the node labeled l_2. (2b) Inserting a node labeled l_2 as the child of the node labeled l_1. (2c) A relabeling of the node label l_1 to l_2.

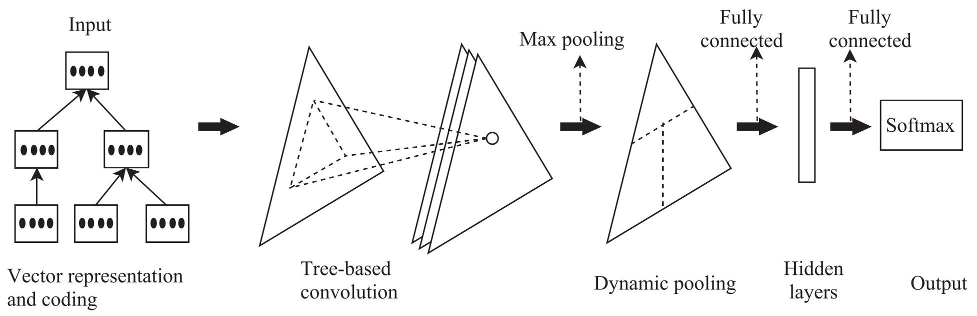

Fig. 3. The architecture of the tree-based convolutional neural network (TBCNN).

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

One problem is that determining the number of weight matrices in Eq. (3) 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 [36]. 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.

3.5. Software metrics-based approaches

Solving software engineering problems brings many benefits to companies and users. For example, estimating software efforts helps managers determine the feasibility of software projects and prepare an appropriate schedule of using resources and budget. Predicting and fixing defects before delivering to customers enhance the reliability and quality of software products. In practice, software defects such as errors, bugs, or failures have led to extremely serious consequences regarding time, finance, and a threat to human well-being. Despite great efforts to improve software quality, the existence of defects in software components is unavoidable. These issues motivate researchers as well as practitioners to develop useful tools to control software projects. The popular approaches are building predictive models from learning software projects' properties called software metrics.

In the software engineering area, software metrics are widely applied to supply engineering and management information and to improve processes, products, and services. Software metrics 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 [22]

To calculate software metrics, we apply a source code metrics and quality analysis tool, namely RSM[2]. Table 1 shows a brief description of 17 file metrics of programs extracted from the RSM tool. Only file metrics are selected because, in the experimental dataset, each program is contained in one file. Then we try various machine learning algorithms to build the predictive models and classify the metrics data using Weka[3]'s implementation.

4. The proposed approaches

In this section, we present our proposed approaches for solving program classification. Firstly, we introduce two pruning

[2] http://msquaredtechnologies.com/m2rsm/
[3] http://www.cs.waikato.ac.nz/ml/weka/

Table 1
List of software metrics of source code files.

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

techniques for reducing the complexity of ASTs. Next, we describe two combination models of TBCNN and kNN, SVM to boost the performance of TBCNN[4]. Our expansion models, AST datasets, and the output data are also publicly[5].

4.1. Data preprocessing

4.1.1. Pruning redundant branches and reconstructing AST

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;
}
```

[4] https://sites.google.com/site/treebasedcnn/
[5] https://github.com/nguyenlab/TBCNN_kNN_SVM.git

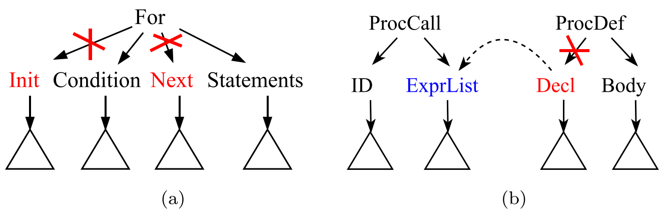

Fig. 4. The operations of pruning redundant branches of `For` and `ProcDef` (Procedure Definition) sub-trees.

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. 4a) 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--;
  }
```

   For above reasons, we remove two uninformative children of ASTs of `For` statements including `Init` and `Next` (Fig. 4a).
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. 4b).
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.

4.1.2. Pruning minor procedure sub-trees

In software engineering, a well-organized program is usually divided into smaller procedures, where each of them performs a specific task; and, the main procedure invokes the other ones during the flow of execution for the program. Inspired by the idea that the function of a program may be shown in the major procedure, which has the biggest number lines of code in the program, we cut down all AST's branches of the other procedures. The following program is an example.

```
int N, A;
int acSearch(int n, int i) {
   int ret = 0;


   if(n == 1) ret = 1;
   else for(; i <= n; ++ i)
      if(n%i == 0) ret += acSearch(n/i, i);


   return ret;
}
```

We can see that the purpose of given program is implied in procedure "acSearch", while the procedure "main" only contains data input and procedure invoke statements.

4.2. The combination model of kNN-TED and TBCNN

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) as follows:

$$DecVal_j^i = (1 - t)*Prob_j^i + t*MF\,(nnDist_j^i) \tag{4}$$

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. (5):

$$L^i = \begin{cases} L_{n_1} & \text{if } L_{n_1} = L_{n_2} = .. = L_{n_k} \\ l & \text{if } DecVal_l^i = max\,\{DecVal_j^i\} \end{cases} \tag{5}$$

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) and (5) 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); and the predicted label is determined according to Eq. (5).

4.3. The integration model of TBCNN and SVM

To make an accurate prediction model, we integrate TBCNN and an SVM classifier. In the model, TBCNN is able to capture underlying meaning inside tree nodes; the SVM classifier is a powerful classifier, which showed 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 [37–39].

Fig. 5 illustrates the architecture of the integration model. It consists of two components: 1) TBCNN extracts semantic features of programs, 2) SVM is adopted to build the classifiers based on extracted features. Specially, TBCNN serves as a supervised approach to learn program vector representations from source code. We 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. 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 model with the tree to generate the vector representation.
- Using the SVM classifier to predict the class label of the vector.

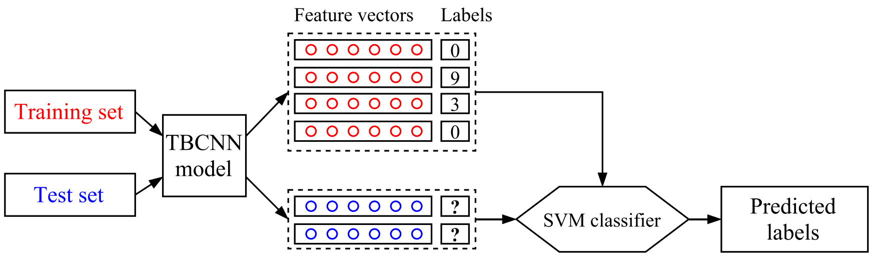

Fig. 5. The integration model of TBCNN and SVM for program classification.

Table 2
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	172.0	102.6
# of AST leaves	90.5	53.8	66.5	47.7	83.4	52.8
Avg. leaf nodes' depth in an AST	7.6	1.7	8.3	1.9	6.7	1.7
# nodes of the smallest AST	29	–	7	–	22	–
# nodes of the largest AST	7,027	–	6,999	–	7,026	–

5. Experimental setup

5.1. The dataset

In this paper, we verify the proposed approaches by solving program classification problem 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 [16]. 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 2 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 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 2, applying the heuristic pruning method reduces the AST nodes notably while the AST nodes decrease slightly in case of pruning minor procedures' branches. The reason for the slight decrease is that many programs contain only one procedure (the `main` procedure). In addition, the average leaf nodes' depth in an AST increases after performing heuristic pruning. This means most of the redundant branches are shorter than the meaningful branches.

5.2. 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. 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 [35], and Levenshtein distance [40]. 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 [41] 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 [42]. To adapt this model for program classification, each program is considered as a document, whereby each statement is equivalent to a sentence. The

Table 3
The performance of classifiers on the 104-label program dataset.

Method	Test accuracy (%)
Tree kernel SVM	58.39
GRNN+LSTM	80.61
kNN + TED	85.84
kNN + LD	83.08
TBCNN	**92.63**

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.

6. Results and discussion

Table 3 shows the classification accuracy on the test set using different approaches. As can be seen, most of the approaches achieve high performance on the experimental dataset. The TBCNN of Mou yields a remarkable accuracy of 92.63% due to subtree feature detectors which have good ability to capture structural information of ASTs. For the kNN algorithm, the accuracies reach 85.84% with TED and 83.08% with LD. Besides, GRNN obtains the accuracy of 80.61% depending on the size of vectors.

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 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 several pruning tree techniques and reconstructing sub-trees (Section 4.1.1, 4.1.2). Table 4 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 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 loss of useful information. Unlikely, pruning minor procedures results in cutting down meaningful information. In this case, although the running time is shortened, the accuracies of the most classifiers are lower than the case of the original dataset (Table 4). The meaningful branches are lost when applying minor procedure pruning because of two reasons: the size is not a good measure to estimate the importance of a procedure, and the function of a program may imply in several its procedures. This also leads to some decrease in strength of classifiers when combining both techniques.

Table 4
Performance comparison of the pruning approaches in terms of accuracy and execution time (ASTs$_{OR}$ are the original ASTs; AST$_{MP}$, and ASTs$_{HP}$ are the ASTs after pruning by minor procedure, heuristic techniques, respectively; and ASTs$_{\mathrm{HP\_MP}}$ are the ASTs after using both pruning techniques.).

Method	ASTs$_{OR}$		ASTs$_{MP}$		ASTs$_{HP}$		ASTs$_{\mathrm{HP\_MP}}$	
	Acc. (%)	Avg. time(s)	Acc. (%)	Avg. time(s)	Acc. (%)	Avg. time(s)	Acc. (%)	Avg. time(s)
kNN + TED	85.84	259.59	84.44	174.03	**86.35**	108.8	84.65	102.9
kNN + LD	83.08	4.78	81.77	4.47	**85.56**	2.39	84.54	2.32
Tree kernel SVM	58.39	0.26	59.57	0.219	**62.65**	0.249	62.45	0.208
GRNN+LSTM	80.61	454.37	80.76	363.48	**83.31**	338.2	80.14	303.18
TBCNN	92.63	1194	91.57	648	**92.88**	810	90.82	439

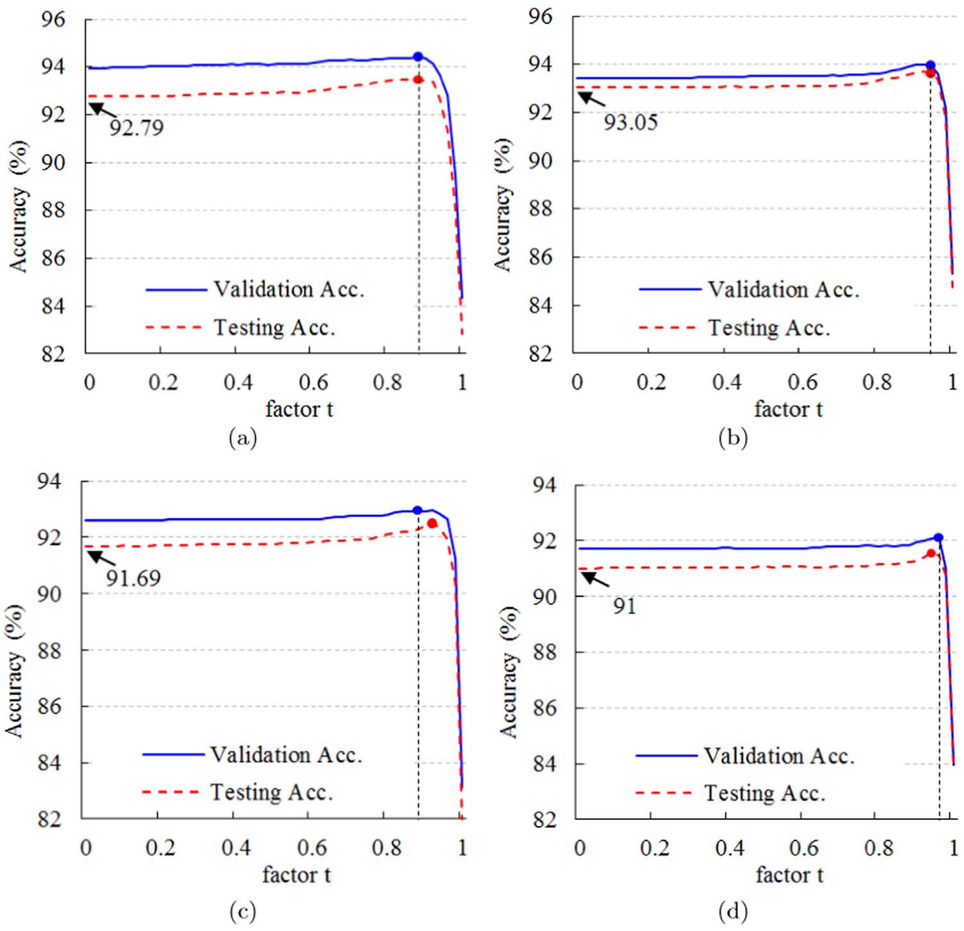

Fig. 6. Tuning the factor t for the combination model of TBCNN and kNN-TED. (6a) before pruning trees, (6b) pruning trees by heuristics, (6c) pruning minor procedures of trees, and (6c) combining two pruning techniques.

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.

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. 6 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), 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. 6 show that at $t = 0$, the accuracies of the hybrid model are higher than those of TBCNN (Table 4). According to the Eq. (5), 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. These mean that although many structures are similar, TBCNN could not detect them. Thus, using extra structural information of ASTs is helpful in making the final decision on class labels of instances. In Fig. 6a and Fig. 6b, the accuracies on the validation sets and the test sets reach maxima at the same values of t. However, in Fig. 6c and Fig. 6d, the accuracies on the validation sets and the test sets reach maxima at different values of t. The inconsistency may be caused by loss of meaningful branches when applying the minor procedure pruning.

The SVM classifier is known as a powerful classifier, which has high accuracy and ability to deal with high-dimensional data. TBCNN is able to capture underlying meaning inside tree nodes. For this reason, we generated the integration model of two such

Table 5
Accuracy of the proposed models in comparison with TBCNN.

Method		ASTs_{OR}	ASTs_{MP}	ASTs_{HP}	$\text{ASTs}_{\text{HP\_MP}}$
TBCNN		92.63	91.57	92.88	90.82
TBCNN+kNN-TED		93.48	92.27	***93.69***	***91.46***
	Kernel				
	Linear	93.42	92.14	93.43	91.40
TBCNN+SVM	Polynomial	91.69	89.83	91.58	88.21
	RBF	**93.74**	**92.53**	**93.89**	**91.81**
	Sigmoid	***93.63***	***92.37***	93.60	91.36

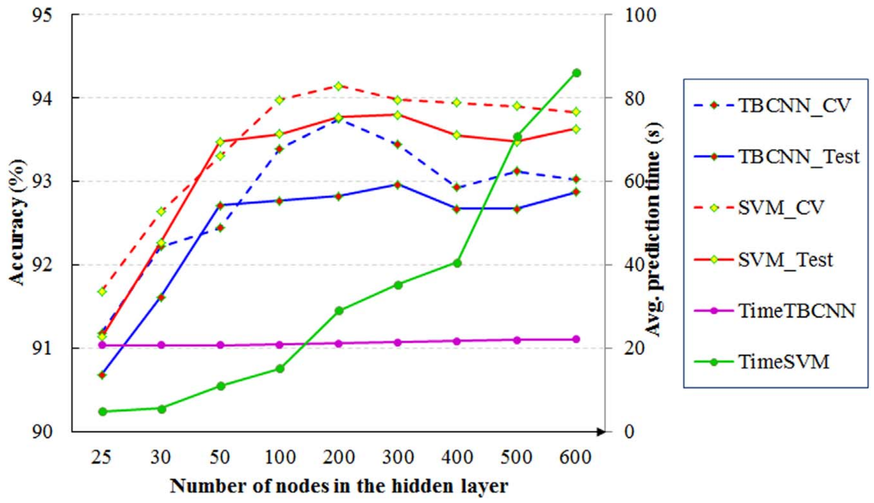

Fig. 7. The accuracies on validation and test sets in the case of heuristic pruning (HP), and the time for predicting an instance of classifiers with different numbers of hidden nodes. TBCNN_CV, TBCNN_Test, SVM_CV and SVM_test show the accuracies of TBCNN and TBCNN-SVM classifiers on validation and test sets, respectively. TimeTBCNN and timeSVM represent the time for predicting an instance using TBCNN and SVM.

components. Table 5 compares our proposed model with TBCNN. Generally, the proposed model improves the accuracy of TBCNN remarkably. The TBCNN + SVM-RBF model outperforms others on all experimental datasets.

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. 7 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.

7. 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. We also present this problem in Section 4.1.1 and 4.1.2. 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.

8. Conclusion

In this paper, 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 6).

Acknowledgments

This work was supported partly by JSPS KAKENHI Grant number 15K16048 and the Ministry of Training and Education (MOET), Vietnam under the project 911.

References

[1] S. Ugurel, R. Krovetz, C.L. 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, ACM, 2002, pp. 632–638.
[2] R. Mo, Y. Cai, R. Kazman, L. Xiao, Q. Feng, Decoupling level: a new metric for architectural maintenance complexity, in: Proceedings of the 38th International Conference on Software Engineering, ACM, 2016, pp. 499–510.
[3] H. Dumitru, M. Gibiec, N. Hariri, J. Cleland-Huang, B. Mobasher, C. Castro-Herrera, M. Mirakhorli, On-demand feature recommendations derived from mining public product descriptions, in: 2011 Proceedings of the 33rd International Conference on Software Engineering (ICSE), IEEE, 2011, pp. 181–190.
[4] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: a proposed framework and novel findings, IEEE Trans. Softw. Eng. 34 (4) (2008) 485–496.
[5] S. Wang, X. Yao, Using class imbalance learning for software defect prediction, IEEE Trans. Reliab. 62 (2) (2013) 434–443.
[6] S.-J. Huang, N.-H. Chiu, L.-W. Chen, Integration of the grey relational analysis with genetic algorithm for software effort estimation, Eur. J. Operat. Res. 188 (3) (2008) 898–909.
[7] J. Kaur, S. Singh, K.S. Kahlon, P. Bassi, Neural network-a novel technique for software effort estimation, Int. J. Comput. Theory Eng 2 (1) (2010) 17.
[8] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, IEEE Trans. Softw. Eng. 33 (1) (2007) 2–13.
[9] C. Jones, Strengths and weaknesses of software metrics, Am. Programmer 10 (1997) 44–49.
[10] F. Akiyama, An example of software system debugging., in: IFIP Congress (1), vol. 71, 1971, pp. 353–359.
[11] M.H. Halstead, Elements of software science, Vol. 7, Elsevier New York, 1977.
[12] J. Sayyad Shirabad, T. Menzies, The PROMISE Repository of Software Engineering Databases., School of Information Technology and Engineering, University of Ottawa, Canada (2005). URL ⟨http://promise.site.uottawa.ca/SERepository⟩.
[13] C. Catal, Software fault prediction: a literature review and current trends, Expert Syst. Appl. 38 (4) (2011) 4626–4636.
[14] M. Allamanis, H. Peng, C. Sutton, A convolutional attention network for extreme summarization of source code, arXiv:1602.03001.
[15] X. Huo, M. Li, Z.-H. Zhou, Learning unified features from natural and programming languages for locating buggy source code.
[16] L. Mou, G. Li, L. Zhang, T. Wang, Z. Jin, Convolutional neural networks over tree structures for programming language processing, in: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, 2016.
[17] D. Binkley, H. Feild, D. Lawrie, M. Pighin, Software fault prediction using language processing, in: Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007, IEEE, 2007, pp. 99–110.
[18] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier, Clone detection using abstract syntax trees, in: Software Maintenance, 1998. Proceedings., International Conference on, IEEE, 1998, pp. 368–377.
[19] T. Kamiya, S. Kusumoto, K. Inoue, Ccfinder: a multilinguistic token-based code clone detection system for large scale source code, IEEE Trans. Softw. Eng. 28 (7) (2002) 654–670.
[20] A.M. Memon, Q. Xie, Studying the fault-detection effectiveness of gui test cases for rapidly evolving software, IEEE Trans. Softw. Eng. 31 (10) (2005) 884–896.
[21] J.A. Jones, M.J. Harrold, Empirical evaluation of the tarantula automatic fault-localization technique, in: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ACM, 2005, pp. 273–282.
[22] S.H. Kan, Metrics and models in software quality engineering, Addison-Wesley Longman Publishing Co., Inc., 2002.
[23] G. Schulmeyer, J.I. McManus, Handbook of software quality assurance, Van Nostrand Reinhold Co., 1987.
[24] M. Alvares, T. Marwala, F.B. de Lima Neto, Application of computational intelligence for source code classification, in: 2014 IEEE Congress on Evolutionary Computation (CEC), IEEE, 2014, pp. 895–902.
[25] P. Lerthathairat, N. Prompoon, An approach for source code classification to enhance maintainability, in: Computer Science and Software Engineering (JCSSE), 2011 Proceedings of the Eighth International Joint Conference on, IEEE, 2011, pp. 319–324.
[26] E. Chandra, P.E. Linda, Class break point determination using ck metrics thresholds, Global journal of computer science and technology 10 (14).
[27] I. Goodfellow, Y. Bengio, A. Courville, Deep learning, book in preparation for MIT Press (2016). URL ⟨http://www.deeplearningbook.org⟩.
[28] D. Binkley, M. 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 of the Third IEEE International Workshop on, IEEE, 2003, pp. 203–212.
[29] R. Komondoor, S. Horwitz, Using slicing to identify duplication in source code, in: International Static Analysis Symposium, Springer, 2001, pp. 40–56.
[30] C. Liu, C. Chen, J. Han, P.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, ACM, 2006, pp. 872–881.
[31] S. Wang, T. Liu, L. Tan, Automatically learning semantic features for defect prediction, in: Proceedings of the 38th International Conference on Software

Engineering, ACM, 2016, pp. 297–308.
[32] P. Bille, A survey on tree edit distance and related problems, Theoretical Comput. Sci. 337 (1) (2005) 217–239.
[33] E.D. Demaine, S. Mozes, B. Rossman, O. Weimann, An optimal decomposition algorithm for tree edit distance, ACM Transa. Algorithms ((TALG)) 6 (1) (2009) 2.
[34] M. Pawlik, N. Augsten, Rted: a robust algorithm for the tree edit distance, Proceedings of the VLDB Endowment 5 (4) (2011) 334-345.
[35] M. Pawlik, N. Augsten, Tree edit distance: robust and memory-efficient, Inform. Syst. 56 (2016) 157–173.
[36] R. Socher, E.H. Huang, J. Pennin, C.D. Manning, A.Y. Ng, Dynamic pooling and unfolding recursive autoencoders for paraphrase detection, in: Advances in Neural Information Processing Systems, 2011, pp. 801–809.
[37] B. Li, N. Chen, J. Wen, X. Jin, Y. Shi, Text categorization system for stock prediction, Int. J. u-and e-Service, Sci. Technol. 8 (2) (2015) 35–44.
[38] R.M.J.S. Bautista, V.J.L. Navata, A.H. Ng, M.T.S. Santos, J.D. Albao, E.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, IEEE, 2015, pp. 1–6.
[39] G.M. Foody, The effect of mis-labeled training data on the accuracy of supervised image classification by svm, in: 2015 IEEE International Geoscience and Remote Sensing Symposium (IGARSS), IEEE, 2015, pp. 4987–4990.
[40] S. Hjelmqvist, Fast, memory efficient levenshtein algorithm (2014).
[41] A. Moschitti, Efficient convolution kernels for dependency and constituent syntactic trees, in: ECML, , vol. 4212, Springer, 2006, pp. 318–329.
[42] D. Tang, B. Qin, T. Liu, Document modeling with gated recurrent neural network for sentiment classification, in: Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, 2015, pp. 1422–1432.