Applied Intelligence (2023) 53:12210–12236
https://doi.org/10.1007/s10489-022-04078-y

Identifying algorithm in program code based on structural features using CNN classification model

Yutaka Watanobe[1] · Md. Mostafizer Rahman[2,3] · Md. Faizul Ibne Amin[2] · Raihan Kabir[2]

Accepted: 8 August 2022 / Published online: 23 September 2022

Abstract

In software, an algorithm is a well-organized sequence of actions that provides the optimal way to complete a task. Algorithmic thinking is also essential to break-down a problem and conceptualize solutions in some steps. The proper selection of an algorithm is pivotal to improve computational performance and software productivity as well as to programming learning. That is, determining a suitable algorithm from a given code is widely relevant in software engineering and programming education. However, both humans and machines find it difficult to identify algorithms from code without any meta-information. This study aims to propose a program code classification model that uses a convolutional neural network (CNN) to classify codes based on the algorithm. First, program codes are transformed into a sequence of structural features (SFs). Second, SFs are transformed into a one-hot binary matrix using several procedures. Third, different structures and hyperparameters of the CNN model are fine-tuned to identify the best model for the code classification task. To do so, 61,614 real-world program codes of different types of algorithms collected from an online judge system are used to train, validate, and evaluate the model. Finally, the experimental results show that the proposed model can identify algorithms and classify program codes with a high percentage of accuracy. The average precision, recall, and F-measure scores of the best CNN model are 95.65%, 95.85%, and 95.70%, respectively, indicating that it outperforms other baseline models.

Keywords Program code classification · Structural features · Algorithm identification · Program code · Programming education · Software engineering · Convolutional neural network

1 Introduction

Information technology (IT) has become an indispensable part of global society. One of the essential requirements for developing IT tools is computer programming and the importance of programming education is attracting global attention [1]. Programming languages, curriculum, teaching, and learning methods as well as platforms have become the subject of representative basic research on programming education [2–4]. As a result, a considerable amount of code[1] is generated and accumulated daily by learners at different levels in platforms such as online judge (OJ) systems [5]. These large code archives can be used as a suitable reference for problem solving, searching for problems and answers as well as for educational research and analysis [6]. In the context of education, identifying the algorithm in the code can be useful for advanced code analysis, including code evaluation [7, 8], plagiarism checking, and problem evaluation (or difficulty estimation) [9, 10]. Furthermore, educational data mining (EDM) using large-scale programming data from repositories enable various empirical analyses. These analyses demonstrate the correlation between academic achievement and programming skills, user assessment, learning path recommendations to facilitate programming learning [1, 11, 12].

Yutaka Watanobe and Md. Mostafizer Rahman are contributed equally to this work and considered primary contributors.

✉ Md. Mostafizer Rahman
mostafiz26@gmail.com

Extended author information available on the last page of the article.

[1]The terms code, solution code, source code, and program code are used interchangeably.

In software engineering (SE), algorithms are implemented at the functional level of the code. Solution codes can be reused for various purposes in SE in the form of libraries, open sources, components and APIs. One of the important aspects for faster coding is code reuse [13]. Code reuse is a practice of using existing code snippets to create a new function or code, as it requires understanding other codes and algorithms used. The identification of algorithms are also important for development environments (IDEs, editors, etc.) and related intelligent software tools, where feedback and support functions are involved. In a development environment, services for various types of searches against a set of program codes are indispensable. Identifying algorithms in code can be useful for advanced code analysis, including code cloning, refactoring, function prediction, debugging, code evaluation, and software metrics. On the other hand, as intelligent software tools, various ML models have been specifically designed for generating, evaluating, modifying, supplementing, and improving source code. The accuracy and efficiency of many specialized ML models for these operations as well as augmentation and retrieval tasks are highly dependent on identifying the program code [14, 15]. Therefore, the algorithm implemented in the code can be a useful feature for ML models.

Due to the vast amount of code accumulated, manually searching for codes using keywords, comments/documents, tags, names, and other metadata is a challenging task. The unavailability, non-uniformity, and inadequacy of metadata is also a major obstacle in code retrieval. This is because many keywords are freely defined by programmers, the main reason for non-uniformity, and these keywords may not be suitable for accurate code classification. However, to find similar codes for reference purposes, it is not enough to find identical codes of similar algorithms based only on metadata. Therefore, artificial intelligence (AI) can be useful as a core technology to solve this problem. In recent years, advanced deep neural network (DNN) models, such as recurrent neural networks (RNN), feed-forward neural networks (FNN), long short-term memory (LSTM) [16], bidirectional long short-term memory (BiLSTM) [17], and convolutional neural network (CNN) [18], are effectively used for such diverse tasks as computer vision [19–22], travel and Internet-of-Things time series data [23, 24], fault diagnosis of chemical data [25], and autonomous transportation systems [26]. Meanwhile, DNN models are considered an effective method in the context of programming activities.

In recent times, DNN models have achieved significant results for program code classification, recommendation, error detection, prediction, and code assessment [7, 12, 27–30]. Moreover, DNN models are used for various programming tasks (e.g., code completion, evaluation, repair, generation, and summarization) [31–33]. To make DNN models more effective in programming-related tasks, real-world programming data resources can be advantageous, and one of the sources can be OJ data. The OJ system is an effective platform for programming exercises and competitions, allowing programmers to practice throughout the year [34, 35]. OJ systems can effectively provide autonomous learning opportunities for code evaluation and detailed feedback on program errors [9, 10, 12]. Let $P = \{p_1, p_2, p_3, \cdots, p_n\}$ be the set of problems related to various algorithms, $V = \{v_1, v_2, v_3, \cdots, v_m\}$ be the set of verdicts. For each problem P, there are many solutions $S = \{s_1, s_2, s_3, \cdots, s_w\}$ and each solution receives a verdict in V with evaluation values such as CPU time. Typically, OJ systems provide decisions or verdicts depending on the errors and acceptance of the codes. Each error decision gives a specific reason for an error in the code. For example, error decisions such as memory limit exceeded (MLE), time limit exceeded (TLE), and runtime error (RE) are made when the performance of the algorithm is not enough for solving the corresponding problem. In contrast, the error decision, e.g., wrong answer (WA), is made when the code contains logical errors. Thus, large real-world OJ data (solution codes with verdict and performance logs for problem sets) can be a real treasure in the task of AI for coding [36–38].

Despite the remarkable results of DNN models in programming tasks, the structural (or algorithmic) features of the code have not been adequately discussed. However, knowing the algorithms used in the program code is important from an educational and software development perspective to better understand the code. Therefore, the classification of program code based on the structural features of the code remains an open problem. To address this research gap, we propose a CNN-based program code classification model that can be applied to both programming education and software development. The proposed model classifies program codes by identifying the algorithms contained in the codes. In addition, this study presents a new data preprocessing approach for program codes. The code preprocessing requires several steps, including (i) user-defined properties/tokens of program codes such as functions, classes, keywords, and variables are filtered; (ii) as the structural features (SFs) such as if, $else$, $loops$, mathematical operators, bitwise operators, and assignment operators of the codes are considered; (iii) the SFs of each program code are converted into a one-hot binary matrix (OBM). We have collected two different datasets of real-world program codes based on various algorithms for model training, validation, and evaluation. Three CNN models are developed, trained, and evaluated based on various structures and hyperparameters to select the best model for program code classification. The best CNN model is applied for the classification task considering the experimental results. The contribution of the research work is as follows:

- The proposed CNN model can identify the algorithm used in the program code and classify the code based on the identified algorithm.
- We present a novel strategy for program code processing. SFs are extracted from program codes and converted into OBM for model training. SFs facilitate the model to understand the algorithmic properties of codes better.
- The average precision, recall, and F-measure values of the proposed model are 95.65%, 95.85% and 95.70%, respectively, which outperform the values obtained by other referenced models.
- The proposed classification model and its novel data preprocessing approach can be useful for various educational and industrial applications.

The remainder of this paper is structured as follows. Section 2 presents the background and related works. Section 3 describes the proposed approach, and Section 4 presents the experimental results and evaluations. Section 5 discusses the results in detail, and finally, Section 6 concludes this study with suggestions for future work.

2 Background and related works

This section presents prior studies related to programming education and its challenges, ML in software development practices and its challenges, code evaluation and repair, and code classification.

2.1 Programming education and its challenges

Research in programming education has gained potential worldwide, and learning programming in higher education has been recognized as significantly important for the sustainable development of IT infrastructure [39]. A data-driven study [1] has shown that better programming skills have a positive impact on students' academic performance. In [40], EDM has been performed to support programming learning based on programming data. Sun et al. [41] have proposed a model to evaluate students' programming skills in terms of programming and test performance. Based on object-oriented programming tasks, the model observed the improvement of students' programming skills. The experimental results showed that test performance was positively correlated with programming performance. Qian et al. [42] conducted a comprehensive study to identify students' misconceptions and difficulties in introductory programming course. Students are most confronted with these misconceptions, such as conceptual, syntactic, and strategic knowledge. The challenges faced by students depend on many factors, including the unusualness of language syntax, programming environments, incorrect concepts and strategies, and instructor competence. Medeiros et al. [2] categorized the challenges in introductory programming and essential issues for learning programming and teaching in higher education. In addition, the study [43] identified significant challenges such as writing, debugging, conceptualizing, and tracing code. To overcome these challenges in learning programming, pedagogical teaching/learning techniques and valuable learning tools are also presented. Meanwhile, due to rapid social and technological changes, many interesting and convenient tools are available, which sometimes have a negative impact on programming learning and students' motivation [39].

2.2 Machine learning in software development practices and its challenges

Recently, ML has been gaining attention as a method for developing various software systems, such as speech recognition, computer vision, natural language processing (NLP), robot control, and other application domains. ML capabilities can be integrated into a software system in many ways, including ML components, tools, libraries (cover ML functionalities), and frameworks [14]. In contrast, a widespread trend has emerged: the development and implementation of ML-enabled systems are fast and inexpensive. However, long-term maintenance is not cost-effective [44]. Wan and collaborators investigated the differences in software development practices between ML and non-ML [14]. Moreover, common practices and workflows for building large-scale ML applications, systems, and platforms at Microsoft, Amazon, and Google have been presented in [6, 45–47]. Additionally, various testing and debugging tools have been proposed to test and debug ML-based applications and systems [48–51]. Despite these efforts, standardization and operationalization of reliable ML systems are inevitable. Based on real-world ML-enabled software development practices [6], around 11 challenges have been identified, from data collection to model evolution, evaluation, and deployment. However, our proposed classification model can be a supporting component in building large-scale ML-based applications and systems dealing with SFs.

2.3 Program code evaluation and repairing

Recently, researchers have made continuous efforts to achieve significant results in this area. Programming languages are quite different from natural languages, as program codes contain a large amount of complex structural information. However, conventional NLP models are inadequate for program codes. Therefore, in [52], a tree-based CNN model for the programming code processing task has been proposed. Rahman et al. [8] presented a model for source code evaluation using LSTM neural networks. These networks have been combined with an attention mechanism to understand the complex context of the code. During code evaluation, the model identified errors, including logic and syntax errors in codes with a high accuracy percentage.

In [53], a multi-modal attention network (MMAN) has been proposed to properly represent the SFs of source codes and improve the reasoning for which features have the most impact on the final results. The MMAN can represent both structured and non-structured features of source codes, using a tree-LSTM for the abstract syntax tree (AST) and a gated graph neural network (GNN) for the control flow graph. In another study [7], an LSTM model has been developed to identify source code errors in C programming. In this model, characters, variables, keywords, tokens, numbers, functions, and classes have been encoded with the defined IDs. The model detected errors in faulty solution codes with high accuracy. Terada et al. [29] presented an interesting model for predicting the following unknown code sequence to complete the code. The model was built using an LSTM network. Their model can help novice programmers who have difficulty writing complete code from scratch. This model has effectively predicted the correct words to complete the code. In addition, code evaluation, completion, and repair tasks have been performed using an LSTM neural network at different levels of programming learning [31, 32].

2.4 Program code classification

The program code classification model is essential for a better understanding of the code. Researchers have proposed various approaches for program code classifications. In the early stages of code classification and prediction, NLP models have been applied to source code to perform various prediction tasks [54–56]. A GNN model [57] was proposed for students' program code classification that integrates AST and data flow to improve the performance of the model. The GNN model classifies student program code with an accuracy of 97%. Fan et al. [28] proposed a method for classifying defective source codes using RNNs with attention mechanisms. Two evaluation indicators, such as area under the curve (AUC) and F1-measure, were used. AUC and F1-measure achieved about 7% and 14% additional accuracy compared to other benchmark models.

Furthermore, many models have been proposed for classifying program codes based on programming languages. Ugurel et al. [58] performed two types of classification using SVM: first, classification of programming languages and, second, classification of different categories of programs (e.g., databases, multimedia, and graphics). Tian et al. [59] used a Latent Dirichlet mapping method to classify the programming language associated with the source code based on the words. Alreshedy et al. [60] presented an ML language model to classify the source code snippets based on the programming language. A multinomial naive bayes (MNB) classifier was used to classify the source code snippets in their works. The contributions of the stack overflow were used as experimental data. This classification method used features such as comments, variables, and functions, instead of syntactic information. Reyes et al. [61] presented a model for classifying source code using LSTM. Archived source codes are classified based on written programming languages. Empirical results show that the LSTM model performed better than the Naive Bayes and linguistic classifier. Gilda has used a CNN model [62] to identify programming languages from source code snippets.

In [63], classification based on code tags has been performed using three classification methods SVM, random forest, and AdaBoost. In [64], the decision tree-based classification method has been used to classify source codes related to sorting algorithms. LeClair et al. [65] mentioned that the source code can be classified into six categories: games, admin, network, words, science, and usage. Xu et al. [66] used LSTM and CNN to identify vulnerabilities in source code. In addition, a CNN-based classification model was used to classify code based on the algorithms used.

In brief, numerous promising methods have been proposed and experimented with in various studies. The researchers have used traditional unsupervised and supervised classifiers. In addition, CNN and LSTM have been employed as language models for source code-related research and applications. However, the relative importance of the methods is challenging to identify. The proposed code classification model differs from other models due to its novel data preprocessing and selection approach for the CNN model. In this study, three CNN models based on different structures and hyperparameters are trained, validated, and evaluated. The best CNN model is selected for the classification task based on the results.

3 Proposed approach

Programmers prefer implementing algorithms for efficient code. However, implementing algorithms in code is not a trivial task. This research aims to identify the algorithm contained in the program code and classify the code based on the identified algorithm. We have used real-world solution codes of different algorithms from programming competitions and academic courses. A crucial step is a data preprocessing for model training and evaluation, where SFs are extracted from the codes, excluding all user-defined elements (e.g., variables, classes, and functions). These SFs of the codes help the DNN model better understand the algorithm's flow. CNN-based classification models are developed for classifying codes with various structures and hyperparameters. Although the CNN models are widely used in computer vision research, they have recently achieved significant success in various programming-related tasks (classification, error detection, prediction, and language modeling) [67, 68]. The proposed classification model includes several phases, from data acquisition to model training and evaluation: (i) data acquisition and categorization, (ii) data preprocessing, (iii) CNN models training, and (iv) program code classification with the optimal CNN model. The basic framework of our proposed approach is shown in Fig. 1. The proposed approach is explained in detail in the following sections.

3.1 Data collection and categorization

Selecting relevant datasets from a real-world data repository is essential in research. In this study, real-world program codes are collected from the Aizu Online Judge (AOJ) system [69, 70]. All program codes are written in the C++ programming language. AOJ is a platform that hosts various academic programming activities and programming competitions. As of February 2022, AOJ has over 3,000 programming problems and 100,000 users. It presents programming problems very efficiently based on categories and algorithms. The AOJ system has archived more than 6 million solution codes and submission logs, creating research opportunities for SE and programming education. For example, IBM and MIT have used solution codes from AOJ for their CodeNet project [36, 71].

In this study, all program codes are divided into two separate datasets: A and B. In Dataset A, we considered the categories that cover a large number of algorithms in computer science and engineering, such as computational geometry problems (CGP), number theory problems (NTP), flow network problems (FNP), shortest path problems (SPP), query for data structures problems (QDSP), and combinatorial optimization problems (COP), as shown in Table 1. These categories include basic algorithms from graph theory, geometry, numerical analysis, puzzles, numbers, search, computational theory, networks, advanced mathematics, and advanced data structures and algorithms. All program codes of each category in Dataset A are collected from the problems of programming competitions in AOJ[2].

[2]https://onlinejudge.u-aizu.ac.jp/challenges/search/categories

As shown in Table 2, all program codes related to sorting, such as counting sort, bubble sort, insertion sort, merge sort, selection sort, shell sort, and quick sort, are contained in Dataset B[3]. In addition, some essential key features such as complexity and method of sorting algorithms are presented.

3.2 Data preprocessing

To achieve better results from DNN models, effective input shapes can play a vital role. It is essential to create a suitable input shape that represents the actual features of the original data. Programming is a highly complex representation than natural languages. Therefore, we extracted suitable features from the program codes for model training so that the model can be trained effectively. The workflow of preprocessing the program code is shown in Fig. 2.

Only structural properties are extracted from the code for tokenization in program code transformation. Usually, program code consists of operators, operands, loops, branches, keywords, methods, and classes. Therefore, key attributes of the program code are extracted. In contrast, user-defined elements such as comments, variables, classes, and functions with little impact, are not considered. A list of featured tokens (T) and their corresponding IDs are shown in Table 3. Initially, SFs are extracted from the program codes according to Algorithm 1. The steps of program code preprocessing are described in the forthcoming subsections.

```
1:  **Define:** Feature Tokens ($\mathcal{T}$) of codes in Table 3
2:  **Input:** Program Codes ($\mathcal{C}$), $\mathcal{C} = \{c_1, c_2, c_3, \cdots, c_n\}$
3:  **Output:** Tokenized SF ($\mathcal{TF}$) for all $\mathcal{C}$
4:  **for** each program code $c_i \in \mathcal{C}$ **do**
5:      **Refine** Code $\mathcal{RC}_i \longleftarrow$ removeComments($c_i$)
6:      **Scan** $\mathcal{RC}_i$ and **Select** $\mathcal{SF}$ in $\mathcal{RC}_i$ where $\forall \mathcal{SF} \in \mathcal{T}$
7:      **Extract** Selected Features ($\mathcal{EF}_i$) $\longleftarrow$ extractSelect-
    edFeatures ($\mathcal{RC}_i$)
8:      **for** each extracted feature $f \in \mathcal{EF}_i$ **do**
9:          $\mathcal{TF}_{f \in \mathcal{EF}_i} \longleftarrow Token\ ID$
10:     **end for**
11: **end for**
12: **Return** $\mathcal{TF}_{c_i \in \mathcal{C}}$
```

Algorithm 1 Extraction of structural features from code.

3.2.1 Comments deletion

All comments in the program code are identified and removed with the $removeComments()$ function because

[3]https://onlinejudge.u-aizu.ac.jp/courses/list

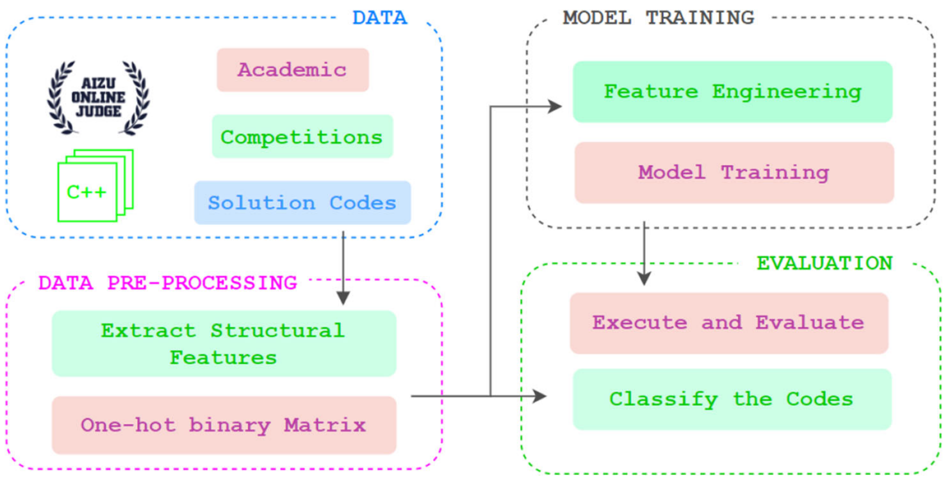


Fig. 1 Framework of the proposed approach

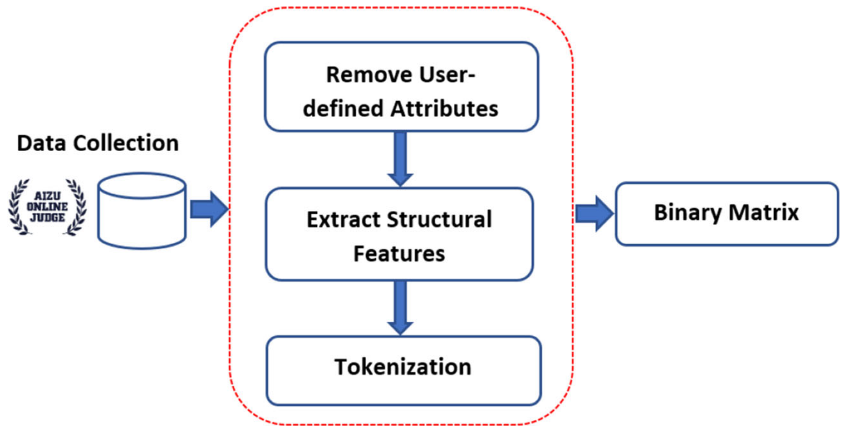
Table 1 Algorithms and number of codes for dataset A

Sl.	Category	Algorithms	# of Codes
1	Computational Geometry Problems (CGP)	Geometric Algorithms, etc.	6299
2	Number Theory Problems (NTP)	modPow, LCM, GCD, Prime Number, Euler's Phi Function, extGCD, etc.	11698
3	Flow Network Problems (FNP)	Maximum flow, Bipartite matching, etc.	1906
4	Query for Data Structures Problems (QDSP)	LCA, Euler Tour, HLD, Union Find, Weighted Union Find, RMQ, RSQ, kDTree, Rolling Hash, Suffix Array, Longest Match, etc.	9917
5	Combinatorial Optimization Problems (COP)	Combinatorial Algorithms, etc.	10222
6	Shortest Path Problems (SPP)	Dijkstra, BFS, Belman Ford, etc.	5356

Table 2 Feature attributes and number of program codes for dataset B

Sl.	Category	Method	Complexity	# of Codes
1	Bubble Sort	Exchanging	$\mathcal{O}(n^2)$	3610
2	Selection Sort	Exchanging	$\mathcal{O}(n^2)$	2918
3	Counting Sort	Counting	$\mathcal{O}(n+k)$	1089
4	Insertion Sort	Insertion	$\mathcal{O}(n^2)$	4092
5	Shell Sort	Insertion	–	1689
6	Merge Sort	Merging	$\mathcal{O}(n \log n)$	1826
7	Quick Sort	Partitioning	$\mathcal{O}(n \log n)$	992

Fig. 2 Workflow of program code preprocessing



the comments in the program code are not significant, as shown in Fig. 3.

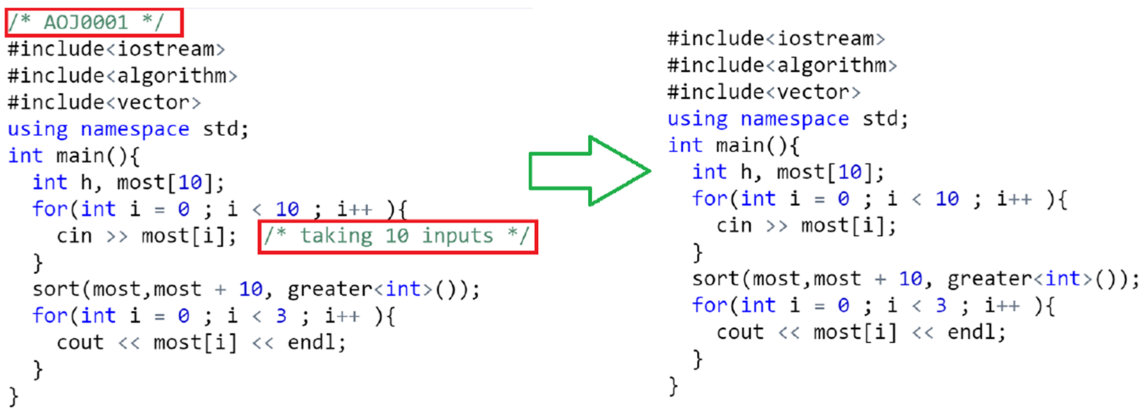
3.2.2 Extraction of feature tokens

After removing comments from the code, feature tokens, such as if, $else$, $loops$, the math operator, bitwise operator, assignment operator, compound assignment operator, comparison operator, braces, parentheses, and square braces, are selected. Typically, in C++ programming, parentheses are used for function calls and declarations, conditional statements (if, $while$, do), $loops$, and operator precedence. In contrast, braces are used for processing functions, classes, structs, if and $loops$. Square brackets are also used to access $arrays$. With this definition, all the feature tokens in the program code are selected for extraction using the $extractSelectedFeatures()$ function, as shown in Fig. 4.

In addition, irrelevant tokens such as functions and variables are identified and removed from the code, e.g., all variables and functions arbitrarily defined by the programmers. The names of the variables and functions may vary depending on the programmer's definition in the code. Thus, a single code can have many different variable and function names. Also, C++ is a statically typed programming language; the type of variables must be explicitly specified in the code, while Ruby and Python are dynamically typed languages. Therefore, all user-defined variable and function names are removed from the code so

Table 3 List of feature tokens and their IDs

Token name	Token Symbol/Sign	Token ID
Assignment Operator	$=$	0
Arithmetic Operators	$+, -, *, /, \%$	1
Bitwise Operators	$\&, \|, \wedge, \sim, \ll, \gg$	2
Compound Assignment Operators	$+=, -=, *=, /=, \%=, ++, --$	3
Compound Bitwise Assignment Operators	$\&=, \|=, \wedge=, \ll=, \gg=$	4
Comparison Operators	$==, !=, <, <=, >, >=$	5
Logical Operators	$\&\&, \|, !$	6
Others	if	7
	$else$	8
	for	9
	$while$	10
	(	11
	)	12
	{	13
	}	14
	[	15
	]	16



```
/* AOJ0001 */
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
int main(){
  int h, most[10];
  for(int i = 0 ; i < 10 ; i++ ){
    cin >> most[i];  /* taking 10 inputs */
  }
  sort(most,most + 10, greater<int>());
  for(int i = 0 ; i < 3 ; i++ ){
    cout << most[i] << endl;
  }
}
```

```
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
int main(){
  int h, most[10];
  for(int i = 0 ; i < 10 ; i++ ){
    cin >> most[i];
  }
  sort(most,most + 10, greater<int>());
  for(int i = 0 ; i < 3 ; i++ ){
    cout << most[i] << endl;
  }
}
```

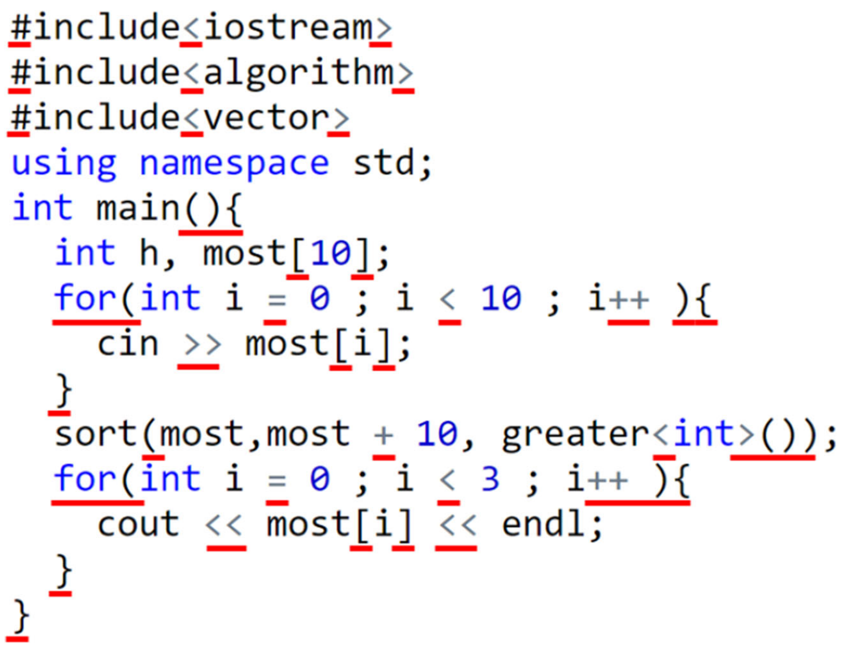
Fig. 3 Identification and removal of unnecessary comments from the code

that the DNN model can better understand the context of the code.

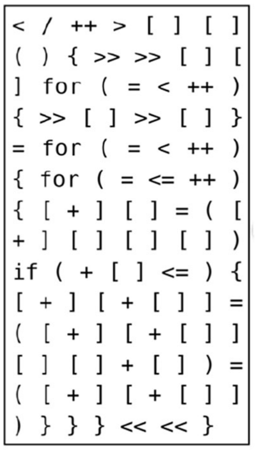
3.2.3 Tokenization of the features

All the feature tokens are extracted from the code, as shown in Fig. 5(a). Next, the extracted feature tokens are converted into token IDs according to Table 3. This process is called tokenization or encoding. In this research, the tokenization/encoding process represents each SF of code as a token. All these tokens are mapped with the numeric numbers to feed DNN models. In learning DNN models, a sequence of tokens is converted into a sequence of numerical vectors, which are later processed by the neural network. Basically, DNN models neither know the SFs such as { + & = [ ] } of the code nor understand the semantic or algorithmic features of the code. Therefore, tokenization/encoding is an important process of DNN to learn the neural network model from scratch. For example, we defined token IDs from 0−16 for different features of the code (Table 3). When some features such as { + & = [ ] } are extracted from the code, these features are converted to numeric numbers such as 13 1 2 0 15 16 14 after tokenization/encoding. Once the tokenization process is completed, the code contains only a list of IDs, as shown in Fig. 5(b).

Fig. 4 Extraction of selected feature tokens

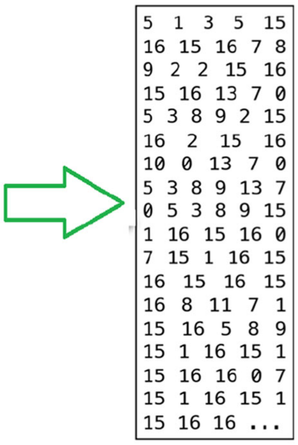


```
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
int main(){
  int h, most[10];
  for(int i = 0 ; i < 10 ; i++ ){
    cin >> most[i];
  }
  sort(most,most + 10, greater<int>());
  for(int i = 0 ; i < 3 ; i++ ){
    cout << most[i] << endl;
  }
}
```



```
< / ++ > [ ] [ ]
( ) { >> >> [ ] [
] for ( = < ++ )
{ >> [ ] >> [ ] }
= for ( = < ++ )
{ for ( = <= ++ )
{ [ + ] [ ] = ( [
+ ] [ ] [ ] [ ] )
if ( + [ ] <= ) {
[ + ] [ + [ ] ] =
( [ + ] [ + [ ] ]
[ ] [ ] + [ ] ) =
( [ + ] [ + [ ] ]
) } } } << << }
```

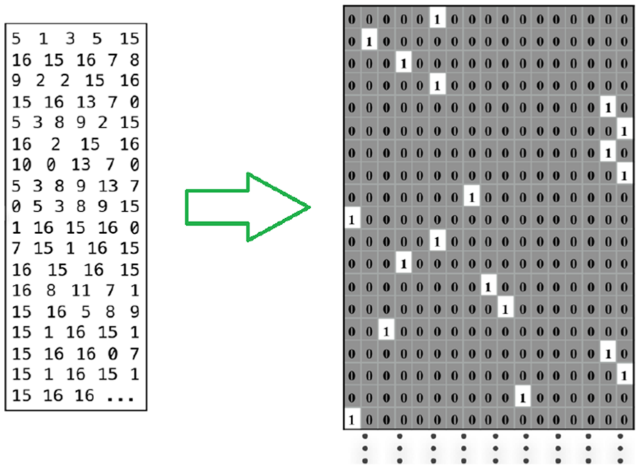
(a)



```
5 1 3 5 15
16 15 16 7 8
9 2 2 15 16
15 16 13 7 0
5 3 8 9 2 15
16 2 15 16
10 0 13 7 0
5 3 8 9 13 7
0 5 3 8 9 15
1 16 15 16 0
7 15 1 16 15
16 15 16 15
16 8 11 7 1
15 16 5 8 9
15 1 16 15 1
15 16 16 0 7
15 1 16 15 1
15 16 16 ...
```

(b)

Fig. 5 (a) A program code with featured tokens, and (b) Tokenization with IDs



```
5 1 3 5 15
16 15 16 7 8
9 2 2 15 16
15 16 13 7 0
5 3 8 9 2 15
16 2 15 16
10 0 13 7 0
5 3 8 9 13 7
0 5 3 8 9 15
1 16 15 16 0
7 15 1 16 15
16 15 16 15
16 8 11 7 1
15 16 5 8 9
15 1 16 15 1
15 16 16 0 7
15 1 16 15 1
15 16 16 ...
```

```
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Fig. 6 OBM conversion process

3.2.4 One-hot binary matrix conversion from token IDs

After the tokenization process, the IDs are assigned to the corresponding feature tokens. A sequence of token IDs is converted to a $\mathcal{P} \times \mathcal{Q}$ matrix structure, where $\mathcal{P}$ represents the number of token IDs and $\mathcal{Q}$ is the highest value of token ID +1. According to the definition of token ID, the maximum value of a token is 16, so the maximum length of the $\mathcal{Q}$ (column of matrices) will be 17. Finally, the token IDs are converted to an OBM structure of $\mathcal{P}$ elements according to Algorithm 2. In Algorithm 2, the concept of the (1) has been used for constructing the OBM. The conversion of token ID to OBM is shown in Fig. 6.

$$\mathcal{M}_{\mathcal{P},\mathcal{Q}} = \begin{cases} 1, & if\, \mathcal{Q} = \mathcal{S}_{\mathcal{P}} + 1 \\ 0, & Otherwise \end{cases} \tag{1}$$

Where $\mathcal{S}_{\mathcal{P}}$ represents the token ID of $\mathcal{P}^{th}$ iteration and $\mathcal{Q}$ is the sequence of column.

In Algorithm 2, first, line 5 takes the entire tokenized solution code (e.g., Fig. 5(b)), then line 6 processes the individual tokens for OBM conversion, and finally lines 7-14 are repeatedly used for OBM construction until the token of a code runs out.

```
1:  **Define:** Maximum Column Number ($MaxC$)
2:  **Input:** Tokenized Structural Features ($\mathcal{TF}$) for all
    program codes $\mathcal{TFC} = \{fc_1, fc_2, fc_3, \cdots, fc_n\}$
3:  **Initialization:** Row $\mathcal{P} \longleftarrow 0$, Column $\mathcal{Q} \longleftarrow 0$
4:  **Output:** OBM for all program codes
5:  **for** each tokenized program code $p_i \in \mathcal{TFC}$ **do**
6:      **for** each token $t \in p_i$ **do**
7:          **while** $\mathcal{Q} \leq MaxC$ **do**
8:              **if** $\mathcal{Q} = t$ **then**
9:                  $\mathcal{M}_{\mathcal{P},\mathcal{Q}} \longleftarrow 1$
10:             **else**
11:                 $\mathcal{M}_{\mathcal{P},\mathcal{Q}} \longleftarrow 0$
12:             **end if**
13:             $\mathcal{Q} \longleftarrow \mathcal{Q} + 1$
14:         **end while**
15:         $\mathcal{P} \longleftarrow \mathcal{P} + 1$
16:         $\mathcal{Q} \longleftarrow 0$
17:     **end for**
18: **end for**
```

Algorithm 2 OBM conversion from tokenized structural feature (TF).

3.2.5 Padding

The final step of preprocessing the program code is padding. This is an essential step for the DNN model with batches. To train a DNN model, all input sequences in each batch must have the same length. Therefore, random tokens are added to the input sequence's end ($post$) and beginning (pre) to make the same length. One of the reasons for this is to avoid overfitting by adding random tokens to the input sequences.

3.3 Architecture of the CNN model

CNN has become an effective deep learning technique for solving complex tasks in various domains in recent years. Thus, the use of CNN has increased significantly in various fields of computer science and engineering [22, 68, 72]. The architecture of a CNN model is illustrated in Fig. 7. The CNN architecture includes different sized convolutional layers (CL), activation functions (AF), max-pooling, fully connected layers (FCL), dropout layers, and softmax function for the classification tasks. The OBM is used as input for different sized CL via a dropout layer. To determine the features of the code for the evaluation process, each CL learned the features of the code from the input sequences. The output of each CL is used as input to the AF (e.g., ReLU/LeakyReLU). The ReLU and LeakyReLU AFs are expressed by (2) and (3), respectively.

$$ReLU\ \ f(z) = max(0, z) \tag{2}$$

$$LeakyReLU\ \ f(z) = max(\alpha z, z) \tag{3}$$

Where z is an input and α is a small magnitude.

A max-pooling layer is added after each CL. The max-pooling layer extracts the maximum value from the output of each activation/feature map generated by the convolutional filter/kernel. In this manner, important information is preserved, and the size of the feature/activation map is reduced. Thereafter, the output of different max-pooling layers is concatenated. The pooled results are passed to an FCL via a dropout layer. The FCL is learned from combining filters that are highly correlated with each algorithm category. Finally, the output of the FCL is converted into probabilities via the softmax layer according to (4). In the following (4), the probability Y_k is calculated from a_k, where a_k is the output of the FCL. The loss function $\mathcal{L}$ is calculated by (5) using the predicted value Y_k and actual value t_k.

$$Y_k = \frac{exp(a_k)}{\sum_{j=1}^{N} exp(a_j)} \tag{4}$$

$$\mathcal{L} = \sum t_k \log(Y_k) \tag{5}$$

The dropout layer was placed in front of CL and FCL to avoid overfitting. The initial dropout layer randomly generates the whole column zero, and the other dropout layers randomly generate some inputs zero. Lastly, the softmax layer is used to classify program codes based on probability. The output probabilities are calculated in the softmax layer for each category based on the given codes. The sum of the probabilities of all the categories is 1 (one). The category with the highest value is declared the winner.

3.4 Hyperparameters

Different architectures and hyperparameters of the CNN model are fine-tuned to select the best/optimal model for program code classification. We used filters/kernels of different sizes such as 16×17, 32×17, and 64×17 in the CL. The lengths/batch sizes ($\mathcal{BS}$) for the input sequences are 16, 32, and 64. However, the horizontal length of the convolutional filter and the OBM is always equal, i.e., 17. The output length of all convolution layers is 64, and thus the length of the training sequence is also 64. The large convolutional filter length is that the CL can learn the characteristics of the entire code block of the program code. Some hyperparameters are given in Table 4.

4 Experimental results

4.1 Overview

In this section, we present the target models and experimental steps, dataset preparation, evaluation metrics, and experimental environment. In this paper, we conducted the experiments in two phases. In the first phase, experiments are conducted using different architectures of CNN models. Based on the performance of the CNN models, the best model is selected for further experiments. In the second phase, experiments are conducted with the best CNN model and two other baseline models (i.e., LSTM and BiLSTM). An overview of the experimental phases is shown in Fig. 8.

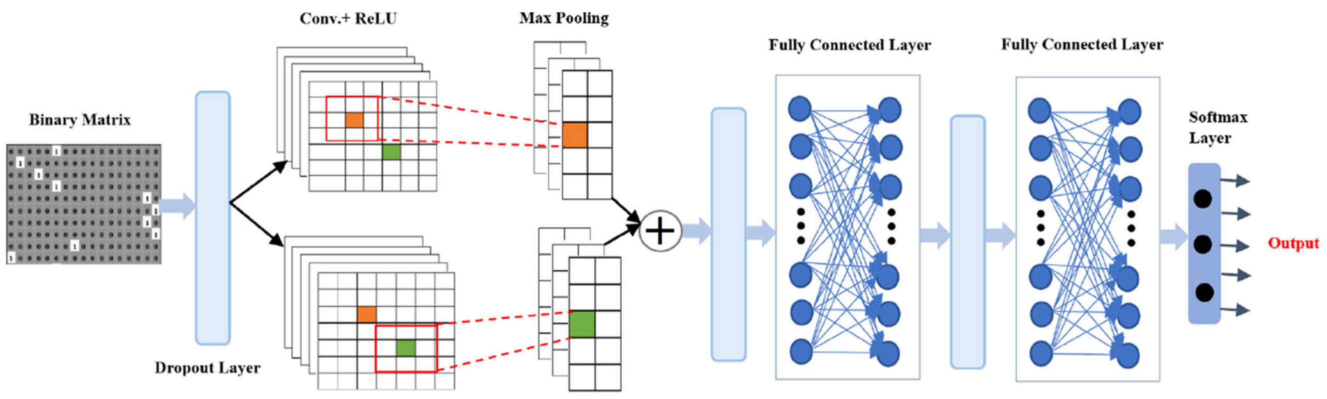


Fig. 7 A sample network architecture of the two-parallel convolutional layered CNN model

Table 4 List of hyperparameters

Name of the parameters	Value
Optimization Algorithm	*Adam*
Learning Rate ($\mathcal{LR}$)	0.01, 0.001, and 0.0001
Batch Size ($\mathcal{BS}$)	16, 32, and 64
Number of iterations for learning	100,000
Epoch	100
Height of convolution filters/kernels	16, 32, and 64
Output length of convolution layers	64
Dropout rate at the first dropout layer	10%
Dropout rate at the second dropout layer	50%
Dropout rate at the third dropout layer	50%

4.1.1 Target models and experimental flow

In this paper, CNN, LSTM and BiLSTM models are used for the training, validation and evaluation purposes. To do so, comprehensive experiments are performed and the main steps are as follows: (i) experiments are performed with different CNN architectures, (ii) training accuracy and time, evaluation accuracy, and comparison are demonstrated, (iii) the best CNN model is selected, (iv) code classification with the best CNN models is performed, (v) code classification with baseline models such as LSTM and BiLSTM is performed, and (vi) comparisons of overall classification performance are made with baseline models and other related studies.

4.1.2 Data preparation for experiments

The details of our datasets and their preprocessing procedures are presented in Sections 3.1 and 3.2. We have two datasets A and B. Dataset A covers a wide variety of algorithms, including combinatorial, geometric, graph, and numerical algorithms. Dataset B, on the other hand, consists of codes related to sorting algorithms. In the experiments, the total number of program codes for datasets A and B of 45,398 and 16,216, respectively, are used, and about 10% of the total number of program codes for each dataset are randomly selected for evaluation. In addition, all the program codes are written in C++ programming language and have been accepted by the AOJ, which means that all codes are "*correct*" and efficient enough. Since dataset A has more program codes and more diversity than dataset B, dataset A is used for training and evaluation in the first phase of the experiment. Next, the best CNN model is selected based on the performance and both datasets A and B are used for evaluation. In the second phase, experiments are performed on dataset A and comparisons are made between the best CNN, LSTM and BiLSTM models.

4.1.3 Evaluation metrics

To evaluate the model performance, precision ($\mathcal{P}_o$), recall ($\mathcal{R}_o$), F-measure ($\mathcal{F}_o$), and accuracy ($\mathcal{A}_o$) are calculated and the following corresponding (6), (7), (8), and (9) are used for these evaluation metrics. A larger $\mathcal{P}_o$ value indicates the higher credibility of the classification results of a particular category. In other words, $\mathcal{P}_o$ indicates the accuracy of classification predictions. The $\mathcal{R}_o$ is an index that measures program codes classified into a certain category; the $\mathcal{F}_o$ is a harmonic mean between $\mathcal{R}_o$ and $\mathcal{P}_o$.

$$\mathcal{P}_o = \frac{TP}{TP + FP} \tag{6}$$

$$\mathcal{R}_o = \frac{TP}{TP + FN} \tag{7}$$

$$\mathcal{F}_o = 2 \times \frac{\mathcal{P}_o \times \mathcal{R}_o}{\mathcal{P}_o + \mathcal{R}_o} \tag{8}$$

$$\mathcal{A}_o = \frac{G}{N} \tag{9}$$

Where TP is the true positive, FP is the false positive, FN is the false negative, G is the correct number of classification and N is the total program codes.

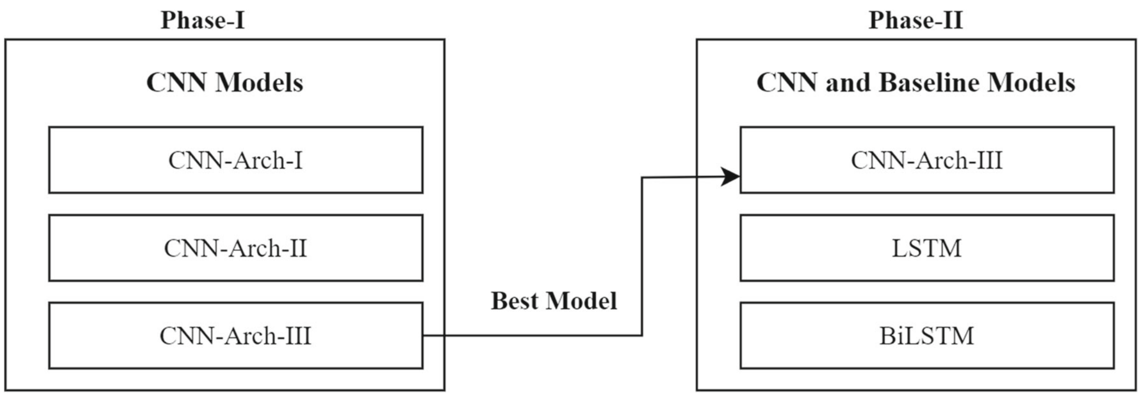


Fig. 8 Overview of the experimental phases

4.1.4 Implementation details

All the experiments are executed on the PyTorch framework with two NVIDIA GeForce GTX 1080 GPU of 32 GB of memory. Details of the experimental hyperparameters of the model are presented in Section 3.4.

4.2 Performance of different CNN models

Instead of using a basic CNN model, we conducted experiments by varying the network architectures and hyperparameters of the CNN to investigate the performance. The optimal CNN model is selected based on the learning accuracy and classification accuracy. For this purpose, three different architectures are developed and used in the experiments: (i) single convolutional layer CNN (CNN-Arch-I) model, (ii) two parallel convolutional layer CNN (CNN-Arch-II) model, and (iii) three parallel convolutional layer CNN (CNN-Arch-III) model. In each CL, three different sizes of filters (e.g., 16, 32, and 64) and two AFs (e.g., ReLU and LeakyReLU) are used separately. In addition, three different BSs (e.g., 16, 32, and 64) and LRs (e.g., 0.01, 0.001, and 0.0001) are used. The network architectures of all three models are shown below.

(i) **CNN-Arch-I Model:** Input (OBM) $\rightarrow$ dropout (10%) $\rightarrow$ Single Conv. Layer[Filter/Kernel Size (64) + ReLU/LeakyReLU + MaxPool] $\rightarrow$ dropout (50%) $\rightarrow$ FCL $\rightarrow$ dropout (50%) $\rightarrow$ FCL $\rightarrow$ Softmax Layer

(ii) **CNN-Arch-II Model:** Input (OBM) $\rightarrow$ dropout (10%) $\rightarrow$ two-parallel Conv. Layer[Filter/Kernel Sizes (32, 64) + ReLU/LeakyReLU + MaxPool] $\rightarrow$ dropout (50%) $\rightarrow$ FCL $\rightarrow$ dropout (50%) $\rightarrow$ FCL $\rightarrow$ Softmax Layer

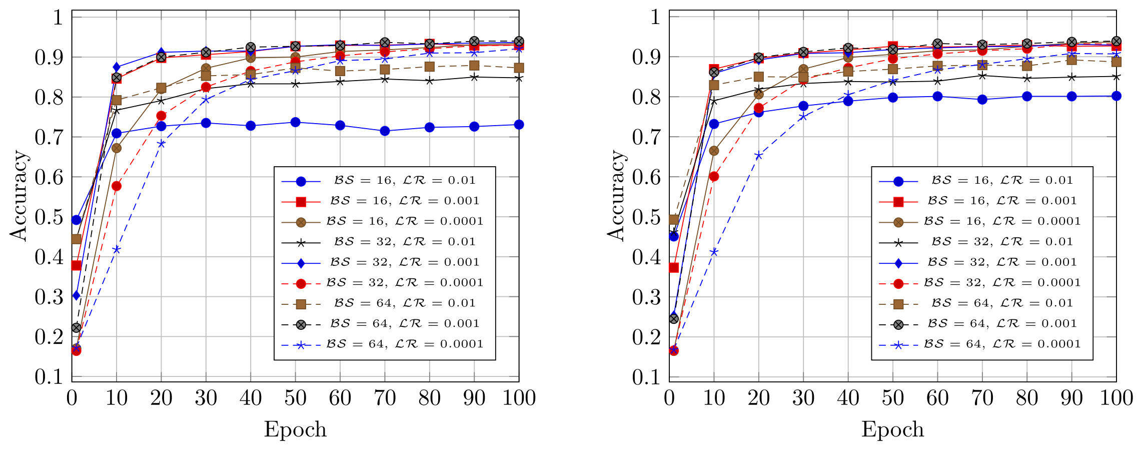
(iii) **CNN-Arch-III Model:** Input (OBM) $\rightarrow$ dropout (10%) $\rightarrow$ three-parallel Conv. Layer [Filter/Kernel Sizes (16, 32, 64) + ReLU/LeakyReLU + MaxPool] $\rightarrow$ dropout (50%) $\rightarrow$ FCL $\rightarrow$ dropout (50%) $\rightarrow$ FCL $\rightarrow$ Softmax Layer

Experimental results of training accuracy and time, classification accuracy, comparisons between training, validation, and evaluation scores, and accuracy with 10-fold cross-validation of the models are presented below.

4.2.1 Training accuracy and time of the CNN models

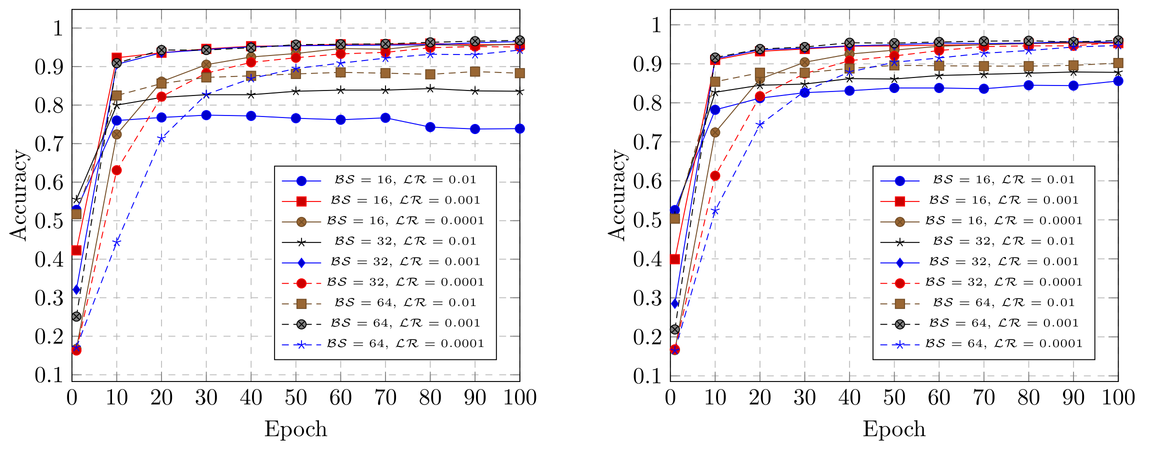
To investigate the training accuracy, Dataset A is first used for model training because it contains a variety of program codes from different algorithms compared to dataset B. Training accuracy is calculated at the 100th epoch of the 1st round of 10-fold cross-validation. Figures 9, 10, and 11 show the training accuracy of the CNN-Arch-I, CNN-Arch-II, and CNN-Arch-III models, respectively. The results reveal that (i) the CNN-Arch-II and CNN-Arch-III models achieved training accuracy of more than 96%, when AF of ReLU/LeakyReLU, $\mathcal{LR}$ of 0.001, and $\mathcal{BS}$ of 16, 32, and 64 are used, as shown in Figs. 10 and 11. (ii) In contrast, the CNN-Arch-I model achieved a training accuracy of approximately 92%, as shown in Fig. 9, which is comparatively lower than the other two models.

The training time for each model with different hyperparameter combinations, i.e., $\{\mathcal{AF}, \mathcal{BS}, \mathcal{LR}\}$, is also compared. Figures 12, 13, and 14 compare the training times for all models, where the X-axis represents $\mathcal{BS}$ and the Y-axis represents the time (in seconds) required to train the model. Several observations can be derived from Figs. 12, 13, and 14: (i) the CNN-Arch-I model required an average



(a) ReLU AF in Conv. Layer (b) LeakyReLU AF in Conv. Layer

Fig. 9 Training accuracy of CNN-Arch-I model



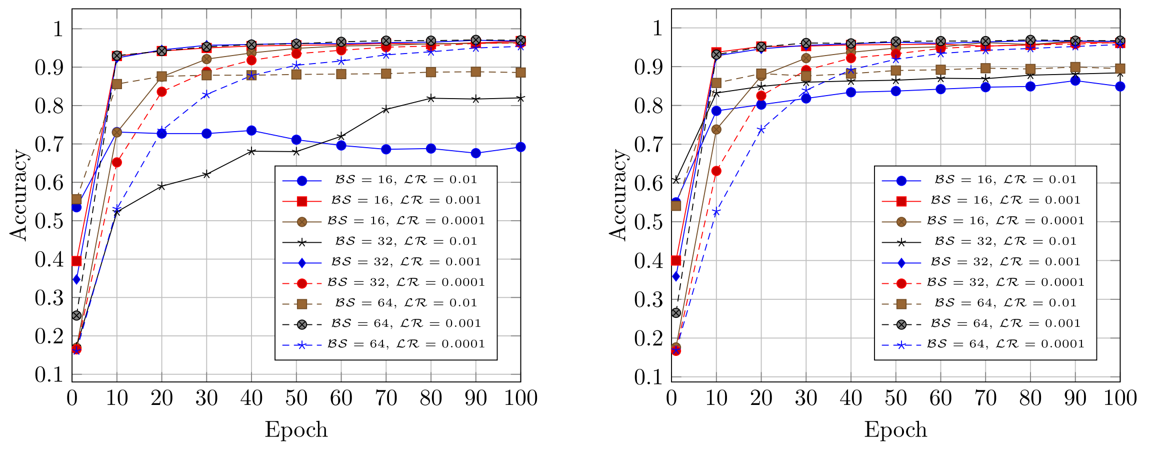
(a) ReLU AF in Conv. Layer (b) LeakyReLU AF in Conv. Layer

Fig. 10 Training accuracy of CNN-Arch-II model

of 26172.77 s (7.27 h) to train for each hyperparameter combination, which is smaller than that required by the other two models. In contrast, the CNN-Arch-II and CNN-Arch-III models required an average of 34772.77 s (9.66 h) and 37595.05 s (10.44 h) to train each hyperparameter combination, respectively; (ii) the training time increased linearly with the increasing number of CLs, regardless of $\mathcal{BS}$ and $\mathcal{LR}$; (iii) all models required relatively more time to train when AF of LeakyReLU was used instead of ReLU. Because ReLU ignores all negative values of neurons, which results in many neurons becoming inactive and generating only 0. This is also known as the dying ReLU/dead neuron problem [73]. LeakyReLU is used to solve the dying ReLU problem by using a small slop value (e.g., $\alpha = 0.01$).

4.2.2 Classification accuracy of the models

In this part of the experiment, we present the initial classification accuracies of all three models with different hyperparameters. All the results are calculated with dataset A on the 100^{th} epoch and the 1^{st} round of 10-fold cross-validation. The top-3 models and their corresponding hyperparameters are selected for further experiments based on the classification results. Tables 5, 6, and 7 show the classification results of the CNN-Arch-I, CNN-Arch-II, and CNN-Arch-III models, respectively. For better understanding, a set of hyperparameters is the combination of different values of $\mathcal{AF}$, $\mathcal{BS}$, and $\mathcal{LR}$, i.e., $\{ReLU,\ 16,\ 0.001\}$. The CNN-Arch-I model achieved the



(a) ReLU AF in Conv. Layer (b) LeakyReLU AF in Conv. Layer

Fig. 11 Training accuracy of CNN-Arch-III model

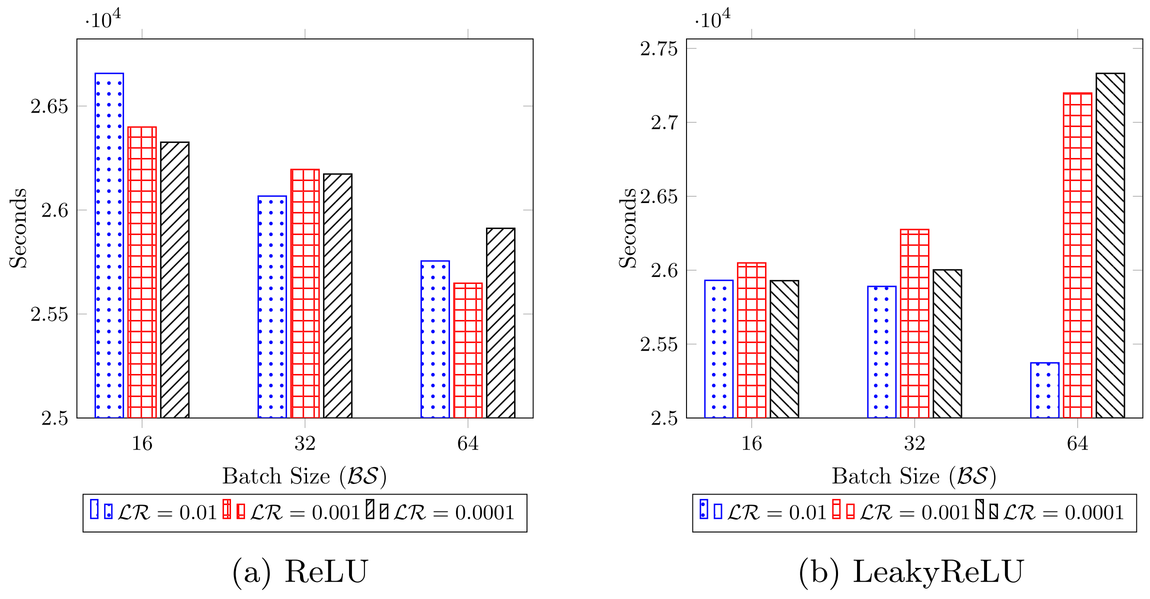


Fig. 12 Training time of CNN-Arch-I model

highest $\mathcal{F}_o$ score of 93.30% and $\mathcal{A}_o$ of 93.10% with this set of hyperparameters $\{LeakyReLU, 32, 0.0001\}$. Similarly, the CNN-Arch-II model obtained the highest $\mathcal{F}_o$ score of 94.20% and $\mathcal{A}_o$ of 93.90% with this set of hyperparameters $\{LeakyReLU, 16, 0.0001\}$. The CNN-Arch-III model achieved highest $\mathcal{F}_o$ score of 94.1% and $\mathcal{A}_o$ of 93.9% with $\{LeakyReLU, 64, 0.0001\}$.

In addition, we performed the experiments with three more layers (e.g., 4, 5, and 6) to demonstrate the performance of the models with the set of hyperparameters $\{LeakyReLU, 16, 0.0001\}$. The CNN model with four (04) convolutional layers (8, 16, 32, 64) achieved $\mathcal{P}_o$, $\mathcal{R}_o$, and $\mathcal{F}_o$ scores of 93.21%, 92.80%, and 92.65%, respectively, and $\mathcal{A}_o$ score of 92.42%. The model required approximately 11 hours and 18 minutes to complete the training process. Similarly, the CNN model with five (05) convolutional layers (8, 16, 32, 64, 128) achieved $\mathcal{P}_o$, $\mathcal{R}_o$, and $\mathcal{F}_o$ scores of 94.03%, 92.10%, and 92.60%, respectively, and $\mathcal{A}_o$ score of 92.17%; the model required approximately 21 hours and 14 minutes to complete the training process. Furthermore, the CNN model with six (06) convolutional layers (8, 16, 32, 64, 128, 256) achieved

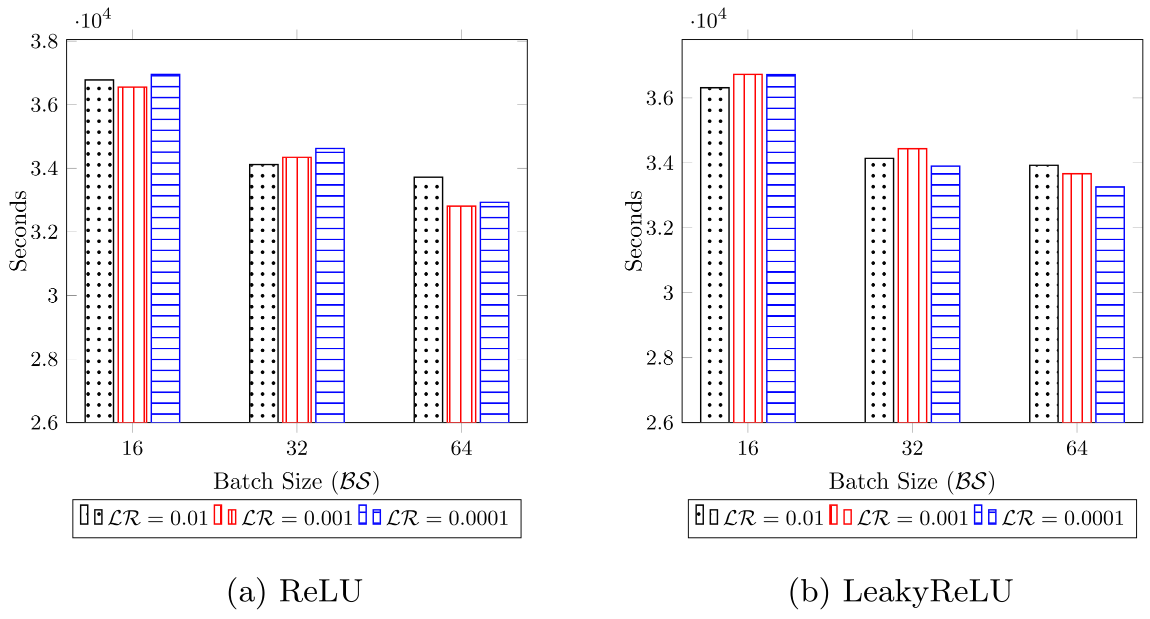


Fig. 13 Training time of CNN-Arch-II model

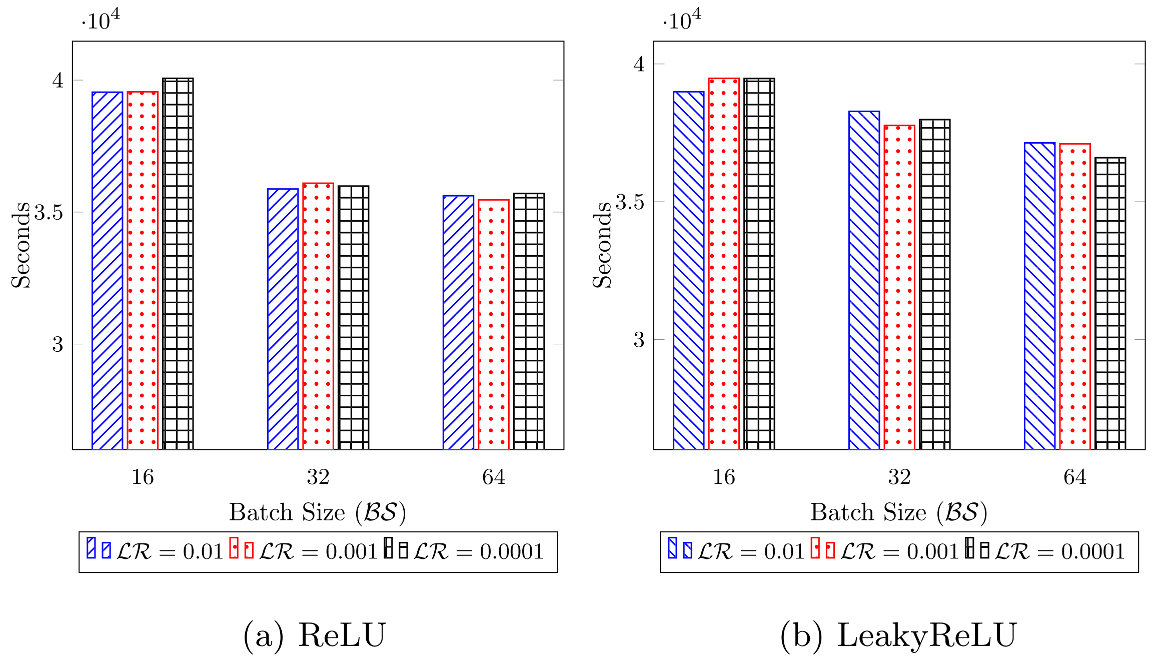


Fig. 14 Training time of CNN-Arch-III model

$\mathcal{P}_o$, $\mathcal{R}_o$, and $\mathcal{F}_o$ scores of 93.31%, 90.93%, and 91.61%, respectively, and $\mathcal{A}_o$ score of 90.90%. The experimental results of these three CNN models (with 4, 5, and 6 convolutional layers) failed to achieve better classification scores and required much more time for training than the CNN-Arch-III model. Depending on the combination of hyperparameters, the models yielded different accuracies. These models achieved a significant classification accuracy

Table 5 Classification results of CNN-Arch-I model

$\mathcal{AF}$	$\mathcal{BS}$	$\mathcal{LR}$	$\mathcal{P}_o$	$\mathcal{R}_o$	$\mathcal{F}_o$	$\mathcal{A}_o$
ReLU	16	0.01	0.828	0.702	0.680	0.618
		0.001	0.926	0.941	0.931	0.929
		0.0001	0.917	0.919	0.915	0.911
	32	0.01	0.875	0.857	0.835	0.808
		0.001	0.915	0.923	0.917	0.914
		0.0001	0.917	0.924	0.918	0.916
	64	0.01	0.858	0.834	0.816	0.792
		0.001	0.905	0.912	0.902	0.901
		0.0001	0.924	0.927	0.925	0.921
LeakyReLU	16	0.01	0.855	0.809	0.814	0.797
		0.001	0.862	0.874	0.857	0.856
		0.0001	0.911	0.901	0.899	0.896
	32	0.01	0.879	0.823	0.836	0.823
		0.001	0.890	0.908	0.894	0.891
		0.0001	**0.931**	**0.937**	**0.933**	**0.931**
	64	0.01	0.886	0.884	0.879	0.871
		0.001	0.939	0.934	0.935	0.929
		0.0001	0.905	0.909	0.904	0.901

Bolded entries in these tables are important in this paper because they are used for comparison and description

Table 6 Classification results of CNN-Arch-II model

$\mathcal{AF}$	$\mathcal{BS}$	$\mathcal{LR}$	$\mathcal{P}_o$	$\mathcal{R}_o$	$\mathcal{F}_o$	$\mathcal{A}_o$
ReLU	16	0.01	0.850	0.591	0.563	0.489
		0.001	0.930	0.902	0.908	0.901
		0.0001	0.931	0.933	0.930	0.929
	32	0.01	0.860	0.810	0.787	0.742
		0.001	0.918	0.924	0.916	0.916
		0.0001	0.938	0.941	0.938	0.936
	64	0.01	0.864	0.862	0.842	0.825
		0.001	0.919	0.925	0.918	0.916
		0.0001	0.918	0.930	0.922	0.921
LeakyReLU	**16**	0.01	0.881	0.807	0.800	0.800
		0.001	0.927	0.921	0.923	0.916
		0.0001	**0.946**	**0.942**	**0.942**	**0.939**
	32	0.01	0.915	0.868	0.879	0.866
		0.001	0.915	0.937	0.928	0.924
		0.0001	0.936	0.944	0.938	0.934
	64	0.01	0.881	0.903	0.881	0.871
		0.001	0.932	0.916	0.920	0.914
		0.0001	0.935	0.933	0.933	0.929

Bolded entries in these tables are important in this paper because they are used for comparison and description

up to about 94%. Therefore, the classification scores of the models help to identify and select the optimal hyperparameters for each model to achieve the best results. Based on the classification $\mathcal{A}_o$ and $\mathcal{F}_o$ scores, the top-3 results are presented in Table 8.

4.2.3 Comparison between training, validation, and evaluation of the top-3 models

To evaluate the performance of the top-3 models, the training, validation, and evaluation curves were compared,

Table 7 Classification results of CNN-Arch-III model

$\mathcal{AF}$	$\mathcal{BS}$	$\mathcal{LR}$	$\mathcal{P}_o$	$\mathcal{R}_o$	$\mathcal{F}_o$	$\mathcal{A}_o$
ReLU	16	0.01	0.795	0.500	0.433	0.376
		0.001	0.936	0.929	0.929	0.921
		0.0001	0.933	0.918	0.920	0.911
	32	0.01	0.852	0.730	0.700	0.643
		0.001	0.896	0.910	0.894	0.896
		0.0001	0.928	0.930	0.926	0.924
	64	0.01	0.874	0.882	0.857	0.838
		0.001	0.940	0.913	0.920	0.911
		0.0001	0.924	0.913	0.915	0.914
LeakyReLU	16	0.01	0.907	0.871	0.880	0.870
		0.001	0.912	0.883	0.885	0.878
		0.0001	0.934	0.937	0.932	0.929
	32	0.01	0.903	0.885	0.889	0.883
		0.001	0.913	0.878	0.880	0.873
		0.0001	0.931	0.936	0.932	0.929
	64	0.01	0.902	0.865	0.864	0.861
		0.001	0.937	0.927	0.929	0.924
		0.0001	**0.940**	**0.943**	**0.941**	**0.939**

Bolded entries in these tables are important in this paper because they are used for comparison and description

Table 8 List of the top-3 results achieved with all three models and hyperparameters

Model	$\mathcal{AF}$	$\mathcal{BS}$	$\mathcal{LR}$	$\mathcal{F}_o$	$\mathcal{A}_o$
CNN-Arch-I	LeakyReLU	32	0.0001	93.30%	93.10%
CNN-Arch-II	LeakyReLU	16	0.0001	94.20%	93.90%
CNN-Arch-III	LeakyReLU	64	0.0001	94.10%	93.90%

which were generated based on 100,000 iterations for the top-3 models, as shown in Fig. 15(a), (b), and (c), respectively. From these figures, the following observations can be made: (i) all models achieved a training accuracy of approximately 96% and a validation and evaluation accuracy of approximately 94%; (ii) during the first 55,000 iterations, all three models experienced more overfitting; (iii) all accuracies increase linearly up to 80,000 iterations and then become more stable.

Although the top-3 models achieved almost similar $\mathcal{F}_o$ score and $\mathcal{A}_o$, as shown in Table 8. Herein, 10-fold cross-validation is performed with the top-3 models and their corresponding hyperparameters to select the best/optimal model. In each cross-validation, different sets of training, validation, and test data are randomly selected to verify the effectiveness of the models. The accuracy comparison between top-3 models for each validation step is shown in Fig. 16. In addition, the average cross-validation accuracy (ACV) is calculated for each model using (10).

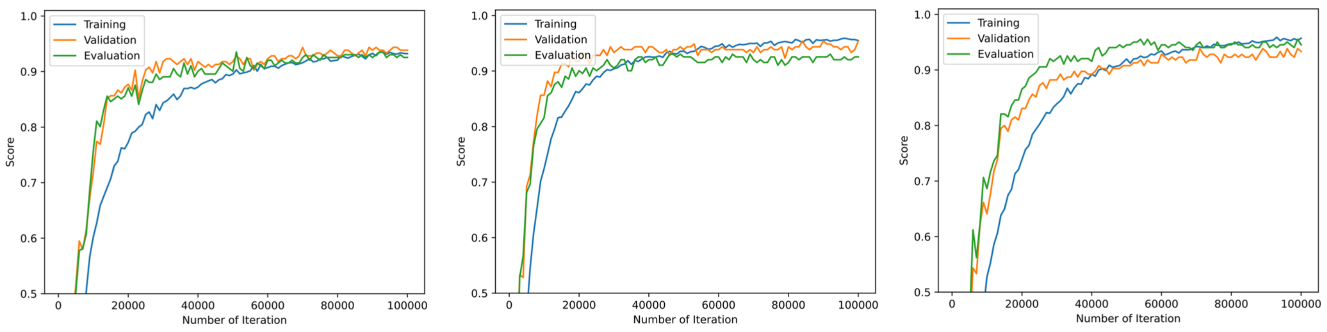
$$ACV = \frac{\sum_{i=1}^{H} A_{o_i}}{H} \tag{10}$$

where H is the number of cross-validation, $\mathcal{A}_o$ is the accuracy.

Figure 16 demonstrates that (i) the CNN-Arch-II and CNN-Arch-III models achieved higher accuracy than the CNN-Arch-I model in 10-fold cross-validations, except for the 10^{th} round of 10-fold cross-validation; (ii) the CNN-Arch-III model achieved an ACV of 92.76%, which is comparatively higher than that of the other two models; (iii) the ACV values of the CNN-Arch-I and CNN-Arch-II models are 91.56% and 92.69%, respectively. Considering the results in training, validation, evaluation, and classification of all the models, the CNN-Arch-III model achieved better results. To validate the superiority of the CNN-Arch-III model, we also performed additional experiments with three (03) more convolution layers (e.g., 4, 5, and 6) (see in Section 4.2.2). The obtained experimental results could not exceed the performance of the CNN-Arch-III model. Henceforth, all experiments are performed with the CNN-Arch-III model.

4.2.4 Effects of tuning hyperparameters

Instead of using a simple CNN architecture, we investigated the performance of CNN models with different architectures and hyperparameters on our dataset. Different sets of hyperparameters, such as $\mathcal{CL} = \{single,\ two-parallel,\ three-parallel\}$, filter/kernel size ($\mathcal{KS}$)= $\{16,\ 32,\ 64\}$, $\mathcal{AF} = \{ReLU,\ LeakyReLU\}$, $\mathcal{BS} = \{16,\ 32,\ 64\}$ and $\mathcal{LR} = \{0.01,\ 0.001,\ 0.0001\}$, were used in three CNN models in different combinations. The performance of each model strongly depends on the hyperparameter settings. For example, CNN-Arch-I has very low accuracy regardless of $\mathcal{AF}$ or $\mathcal{BS}$ when $\mathcal{LR}$ is high (i.e., 0.01). Similarly, the performance of CNN-Arch-II and CNN-Arch-III varies based on changes in hyperparameters. As shown in Figs. 9, 10, and 11, the CNN-Arch-III model achieved the highest training accuracy of approximately 96.7% at the 100^{th} epoch when $\mathcal{AF}$, $\mathcal{BS}$, and $\mathcal{LR}$ were $LeakyReLU$, 64, and 0.001, respectively. As shown in



(a) CNN-Arch-I Model (b) CNN-Arch-II Model (c) CNN-Arch-III Model

Fig. 15 Comparison between training, validation and evaluation accuracy of the top-3 models

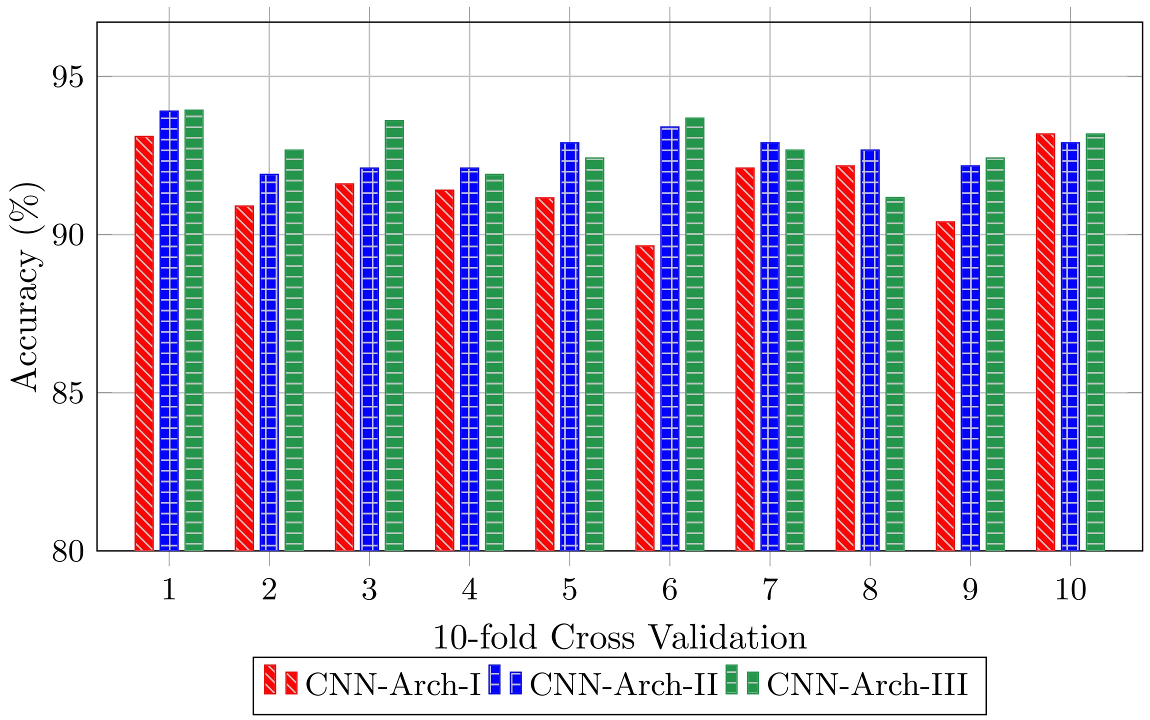


Fig. 16 10-fold cross-validation accuracy of top-3 models

Figs. 12, 13, and 14, the training time increased for all models, regardless of $\mathcal{LR}$ or $\mathcal{AF}$, when $\mathcal{BS}$ was set to 16. In addition, all models consumed approximately 0.60% additional time for training when $LeakyReLU$ is used. Furthermore, in most cases, the models achieved better classification results with $\mathcal{F}_o$ scores of 94.20% and $\mathcal{A}_o$ of 93.90% when $\mathcal{LR}$ was slowed to 0.0001 and $\mathcal{AF}$ was $LeakyReLU$, as shown in Tables 5, 6, and 7. Thus, it can be seen that the optimal hyperparameter settings have a significant impact on model performance.

4.3 Program code classification with the optimal CNN model

In this part of the experiment, the results of program code classification using the best model are presented. The best CNN model (CNN-Arch-III) is used for further experiments. Here, program code classification tasks are performed with datasets (A and B), and the corresponding results are presented.

4.3.1 Model performance with Dataset A

Dataset A contains a large number of program codes on various algorithms such as a tree, graph, geometry, computational theory, discrete mathematics, and data structure (see in Table 1). Therefore, dataset A is more diverse than dataset B. During model training, $\mathcal{P}_o$, $\mathcal{R}_o$, and $\mathcal{F}_o$ scores are calculated using the validation data for each category, as shown in Fig. 17. All learning curves are generated with 100,000 iterations. From the Fig. 17, it is evident that (i) the model obtained relatively low $\mathcal{P}_o$ values (approximately 90%) for the QDSP and COP categories, implying that a large number of false positive (FP) occurred for both categories. On the other hand, about 99% and 96% of $\mathcal{P}_o$ values are obtained for the SPP and CGP categories; (ii) the model achieved about 100% of $\mathcal{R}_o$ values for the FNP category; in contrast, the model scored the lowest $\mathcal{R}_o$ scores for both NTP and SPP categories, indicating that more false negative (FN) occurred; (iii) the value of the $\mathcal{F}_o$ in each category is gradually increased with successive iterations; (iv) the model achieved the highest $\mathcal{F}_o$ values of approximately 98% for both CGP and FNP categories and the $\mathcal{F}_o$ values of these two categories are more stable compared to those in the other categories.

The confusion matrix of $\mathcal{P}_o$ and $\mathcal{R}_o$ is calculated using test data to find out how the model learned the features of program codes of different algorithms. The test program codes for each category COP, CGP, FNP, NTP, QDSP, and SSP are 61, 30, 22, 22, 31, and 35. The confusion matrix of $\mathcal{P}_o$ and $\mathcal{R}_o$ for program code classification is shown in Fig. 18(a) and (b). The confusion matrices indicate that the model achieves approximately 100% of $\mathcal{P}_o$ and $\mathcal{R}_o$ values for the FNP and CGP categories, respectively.

Table 9 shows the validation performance for each category of algorithms. The model (CNN-Arch-III) achieved

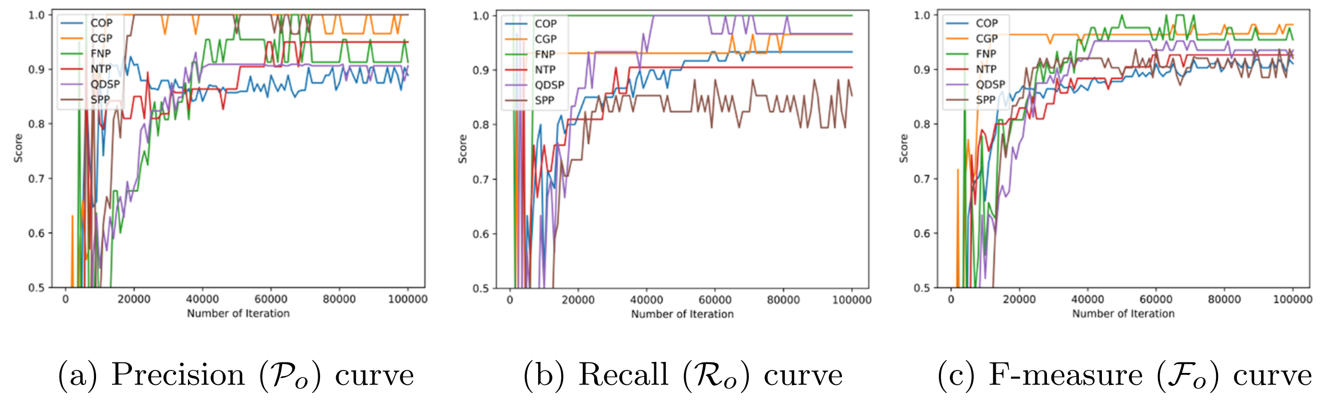


Fig. 17 Precision, Recall, and F-measure curves for each category of algorithm

an average $\mathcal{P}_o$, $\mathcal{R}_o$, and $\mathcal{F}_o$ score of 95.50%, 94.80%, and 95.00%, respectively, for the validation. In contrast, the model achieved an average $\mathcal{P}_o$ score of 94.30%, $\mathcal{R}_o$ score of 94.80%, and $\mathcal{F}_o$ score of 94.50% at the time of evaluation, as shown in Table 10. In this part of the experiment, precision and recall scores are calculated for each category, and validation and evaluation are also performed for dataset A using the CNN-Arch-III model. Given the diversity of dataset A, the overall classification results achieved with the best model are significant.

4.3.2 Model performance with Dataset B

Dataset B is also used for training, validation, and evaluation of the model (CNN-Arch-III) similar to Dataset A. The program codes of Dataset B refer to sorting algorithms. The purpose of all sorting algorithms is the same, but the way they are applied in the codes is different. The SFs of the program codes of all the sorting algorithms are used for model training, allowing the models to learn the actual features of the sorting algorithms, instead of the codes. For the evaluation, the average $\mathcal{P}_o$, $\mathcal{R}_o$, and $\mathcal{F}_o$ are calculated for each category of sorting algorithm, as shown in Table 11. The model obtained an average $\mathcal{P}_o$, $\mathcal{R}_o$, and $\mathcal{F}_o$ scores of 97.00%, 96.90%, and 96.90%, respectively.

Comparing the performance of the model in datasets A and B, the model achieved a higher $\mathcal{F}_o$ score of 96.90% for Dataset B than for Dataset A ($\mathcal{F}_o$ score of 94.50%). This is due to the more diversity of program codes and algorithms in Dataset A so that the model could better process and learn the features of the sorting algorithms.

4.4 Program code classification with the LSTM and BiLSTM models

To compare the classification performance of the proposed model, experiments with baseline models, such as LSTM and BiLSTM are performed under considering the same

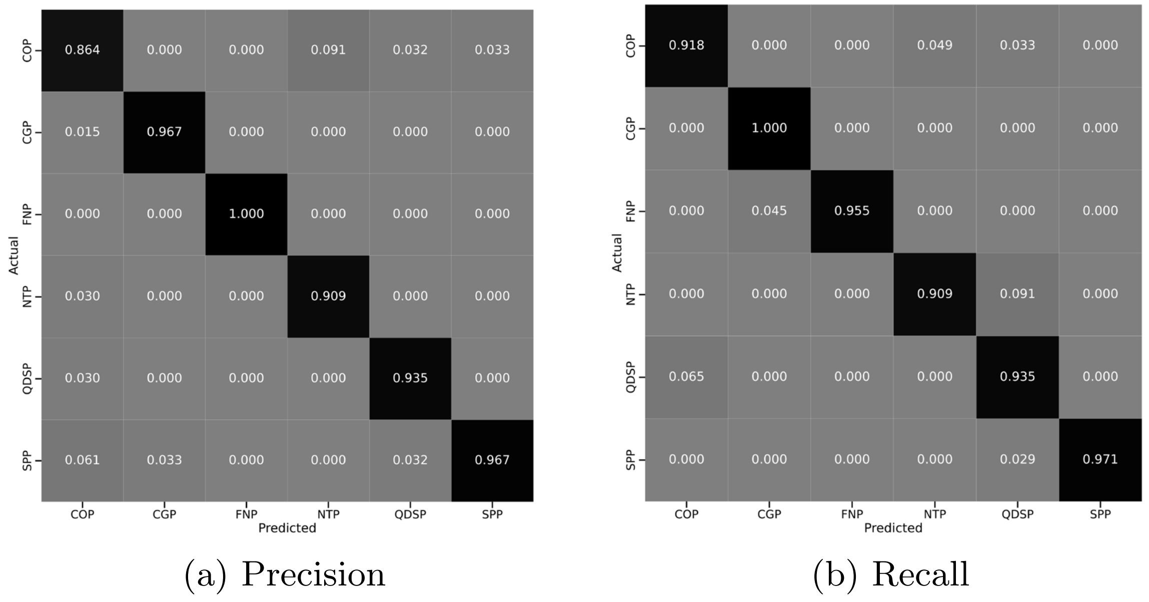


Fig. 18 Confusion matrix of precision ($\mathcal{P}_o$) and recall ($\mathcal{R}_o$)

Table 9 Validation scores for each category of algorithm

Category	$\mathcal{P}_o$	$\mathcal{R}_o$	$\mathcal{F}_o$
COP	0.903	0.933	0.918
CGP	1.000	0.966	0.982
FNP	1.000	1.000	1.000
NTP	0.95	0.905	0.927
QDSP	0.909	1.000	0.952
SPP	0.968	0.882	0.923
Average	**0.955**	**0.948**	**0.950**

Bolded entries in these tables are important in this paper because they are used for comparison and description

dataset. All results are computed at the 100^{th} epoch, and for each combination of hyperparameters, an average of 3480.16 s is required for the LSTM model training. The LSTM model achieved the highest $\mathcal{F}_o$ score of 82.02% and $\mathcal{A}_o$ score of 83.10% with a set of hyperparameters $\{LeakyReLU,\ 16,\ 0.001\}$ as shown in Table 12. Furthermore, the experimental results show that the LSTM model fails to yield significant $\mathcal{A}_o$ and $\mathcal{F}_o$ scores when using the $\mathcal{AF}$ of $ReLU$ and any combination of $\mathcal{BS}$ and $\mathcal{LR}$.

On the other hand, the BiLSTM model achieved the highest $\mathcal{F}_o$ score of 84.14% and $\mathcal{A}_o$ score of 84.64% with the set of hyperparameters $\{LeakyReLU,\ 32,\ 0.001\}$ as shown in Table 13. For each combination of hyperparameters, this model took an average of 6084.83 s to train. In addition, the BiLSTM model achieved relatively good $\mathcal{A}_o$ and $\mathcal{F}_o$ scores for each combination of hyperparameters. Overall, the BiLSTM model achieved comparatively better results than the LSTM model because the BiLSTM model can consider code sequences from both directions (forward and backward), leading to better performance of the model.

4.5 Comparison with baseline models

To validate the effectiveness of our CNN-based program code classification model, different state-of-the-art models that use real-world program codes are compared. The overall approaches, datasets, data preprocessing, model training, validation, and evaluation vary per model. Therefore, two comparisons are made: first, a comparison of the results with the most similar tasks in different studies, as shown in Table 14, and second, a comparison of the results with state-of-the-art models, as shown in Table 15.

Table 10 Evaluation scores for each category of algorithm

Category	$\mathcal{P}_o$	$\mathcal{R}_o$	$\mathcal{F}_o$
COP	0.966	0.918	0.941
CGP	0.968	1.000	0.984
FNP	1.000	0.955	0.977
NTP	0.870	0.909	0.889
QDSP	0.853	0.935	0.892
SPP	1.000	0.971	0.986
Average	**0.943**	**0.948**	**0.945**

Bolded entries in these tables are important in this paper because they are used for comparison and description

Table 11 Evaluation scores for each category of the sorting algorithm (Dataset B)

Category	$\mathcal{P}_o$	$\mathcal{R}_o$	$\mathcal{F}_o$
Bubble Sort	0.986	0.986	0.986
Counting Sort	0.981	0.989	0.984
Insertion Sort	0.990	0.989	0.989
Merge Sort	0.951	0.946	0.948
Selection Sort	0.991	0.992	0.991
Shell Sort	0.955	0.960	0.957
Quick Sort	0.937	0.921	0.928
Average	**0.970**	**0.969**	**0.969**

Bolded entries in these tables are important in this paper because they are used for comparison and description

The experimental results, datasets, number of program codes, languages, and models are considered when making comparisons with other studies, as shown in Table 14. Models such as DP-ARNN [28], RF [28], LSTM [8], and LSTM-AttM [8] are used to classify the defective source codes as either defective or non-defective (i.e., binary classification). In the binary classification, the LSTM-AttM model achieved a comparatively higher $\mathcal{F}_o$ score of 94.00% than the other referenced models. The Stacked Bi-LSTM model achieved an $\mathcal{F}_o$ score of about 89.24% for the multiclass classification task, which is higher than that for other models. In contrast, the proposed CNN-Arch-III model achieved a higher $\mathcal{F}_o$ score of 95.70% than the other comparative multiclass classification models. In addition, the CNN-Arch-III model achieved a higher $\mathcal{F}_o$ score among all classification models (binary and multiclass). Moreover, the experimental data size of our study is 61,614, which is also larger and more diverse than that of the other compared baseline classification models from different studies.

In addition, experiments are performed on the same dataset for LSTM and BiLSTM models, as shown in Tables 12 and 13, respectively. The results are compared with the proposed CNN models, as shown in Table 15. The LSTM model achieved the $\mathcal{F}_o$ score of 82.02% and $\mathcal{A}_o$ score of 83.10%, which are the lowest among all the models, and BiLSTM model obtained $\mathcal{F}_o$ and $\mathcal{A}_o$ scores of 84.14% and 84.64%, respectively, which is better than the LSTM model. The CNN-Arch-III model achieved $\mathcal{F}_o$ and $\mathcal{A}_o$ scores of 94.10% and 93.90%, respectively, which is

Table 12 Classification results of the LSTM model

$\mathcal{AF}$	$\mathcal{BS}$	$\mathcal{LR}$	$\mathcal{P}_o$	$\mathcal{R}_o$	$\mathcal{F}_o$	$\mathcal{A}_o$
ReLU	16	0.01	0.0374	0.1666	0.0612	0.2249
		0.001	0.0436	0.1666	0.0692	0.2620
		0.0001	0.8319	0.8212	0.8258	0.8295
	32	0.01	0.0436	0.1666	0.0692	0.2620
		0.001	0.0436	0.1666	0.0692	0.2620
		0.0001	0.0436	0.1666	0.0692	0.2620
	64	0.01	0.0436	0.1666	0.0692	0.2620
		0.001	0.0436	0.1666	0.0692	0.2620
		0.0001	0.0436	0.1666	0.0692	0.2620
LeakyReLU	**16**	0.01	0.6345	0.6448	0.6248	0.6747
		0.001	**0.8241**	**0.8178**	**0.8202**	**0.8310**
		0.0001	0.8083	0.7964	0.8018	0.8172
	32	0.01	0.0436	0.1666	0.0692	0.2620
		0.001	0.0436	0.1666	0.0692	0.2620
		0.0001	0.0436	0.1666	0.0692	0.2620
	64	0.01	0.0436	0.1666	0.0692	0.2620
		0.001	0.0446	0.1666	0.0704	0.2680
		0.0001	0.6991	0.6911	0.6932	0.7335

Bolded entries in these tables are important in this paper because they are used for comparison and description

better than both LSTM and BiLSTM models. In particular, all CNN models achieved relatively better results than LSTM and BiLSTM models when using the same dataset and hyperparameters. This comparison demonstrates the superiority of the proposed CNN model in understanding the algorithmic features (or SFs) of the code. The overall

Table 13 Classification results of the BiLSTM model

$\mathcal{AF}$	$\mathcal{BS}$	$\mathcal{LR}$	$\mathcal{P}_o$	$\mathcal{R}_o$	$\mathcal{F}_o$	$\mathcal{A}_o$
ReLU	16	0.01	0.6778	0.6702	0.6690	0.7055
		0.001	0.8439	0.8362	0.8398	0.8417
		0.0001	0.8327	0.8350	0.8333	0.8376
	32	0.01	0.7518	0.7564	0.7530	0.7650
		0.001	0.8425	0.8395	0.8406	0.8414
		0.0001	0.8326	0.8332	0.8320	0.8348
	64	0.01	0.7913	0.7490	0.7653	0.7741
		0.001	0.8398	0.8383	0.8384	0.8339
		0.0001	0.8334	0.8396	0.8355	0.8392
LeakyReLU	16	0.01	0.7167	0.6785	0.6873	0.7049
		0.001	0.8393	0.8280	0.8331	0.8364
		0.0001	0.8361	0.8222	0.8284	0.8326
	32	0.01	0.7307	0.6936	0.7069	0.7304
		0.001	**0.8440**	**0.8393**	**0.8414**	**0.8464**
		0.0001	0.8316	0.8304	0.8301	0.8348
	64	0.01	0.7735	0.7463	0.7553	0.7694
		0.001	0.8420	0.8398	0.8406	0.8417
		0.0001	0.8341	0.8258	0.8297	0.8310

Bolded entries in these tables are important in this paper because they are used for comparison and description

Table 14 Experimental results comparison with baseline models of different studies

Models	# of Codes	Language	Classification type	Brief description	$\mathcal{F}_o$ (%)
RF [28]	3,391	Java	Binary	Defective code classification	49.40
LSTM [8]	10,362	C	Binary	Erroneous (syntax and logic) solution code classification	79.00
LSTM [74]	35,000	Multilingual	Multi-class	Classification of codes based on category	81.16
LSTM-AttM [8]	10,362	C	Binary	Erroneous (syntax and logic) solution code classification	94.00
Bi-LSTM [74]	35,000	Multilingual	Multi-class	Classification of codes based on category	83.48
DP-ARNN [28]	3,391	Java	Binary	Defective code classification	56.40
Stacked Bi-LSTM [74]	35,000	Multilingual	Multi-class	Classification of codes based on category	89.24
CNN-Arch-III (ours)	61,614	C++	Multi-class	Algorithm detection in program codes using structural features	95.70

classification results of the proposed CNN models showed the potential for detecting algorithms in program codes.

5 Discussion

In this section, we discussed the approach, including scalability of the model compared to other state-of-the-art models, and usefulness of the model in learning programming and software engineering. In addition, we discuss the threats to the validity of the proposed model.

5.1 Model performance analysis

In this paper, we focus on training the DNN models using the algorithmic features of the code rather than the meta-information. We considered SFs as key components of the algorithm in each solution code. A large number of practice-oriented solution codes are collected and processed for training and evaluation of the model. We conducted extensive experiments with different CNN architectures and hyperparameters. The CNN-Arch-III model achieved better training, validation, and evaluation accuracy than other CNN models. Comparisons are also made between CNN, LSTM, and BiLSTM models to demonstrate the classification performance of these models. Experimental results show that DNN models recognize the algorithm in solution codes with an acceptable degree of accuracy. The CNN-Arch-III model achieved an average $\mathcal{F}_o$ score of 94.5% and 96.9% for datasets A and B, respectively for code classification. This result shows that the model achieved high accuracy in classifying "*program codes*" without meta-information.

In addition, we reviewed a large body of literature on program code classification. We found that studies classify codes based on various types of meta-information of the code, including programming language [58–61], code tags [63], errors [8, 28], and category [64, 65]. To the best of our knowledge, no study has considered the algorithmic (structural) features of codes in the classification task. Consequently, a comparison of the proposed CNN-Arch-III model with other relevant classification methods is presented in Table 14. However, in this paper, we recognize the importance of the algorithmic (structural) features of the codes for the classification task. The experimental results (Tables 5, 6, 7, 12, and 13) show that DNN models have achieved significant results using the SFs of the program codes for the classification task.

5.2 Model scalability

In this study, SFs are extracted from the codes and then the CNN model is trained to classify the program codes. The model classifies the program codes based on the category of algorithms with a high percentage of $\mathcal{F}_o$ score of about 95.7%. The higher accuracy demonstrates that the proposed approach including SFs extraction, OBM conversion, and training and evaluation of the best CNN model with real-world program codes, are effective. Moreover, the experiments are conducted with program codes of C++ programming, which is considered a procedural programming language. Thus, the proposed model can also be utilized for classifying program codes of other procedural languages, such as Python, Java, and C. Based on the comparison studies with the baseline classification models, the proposed model (CNN-Arch-III)

Table 15 Experimental result comparison with baseline models

Models	$\mathcal{P}_o$(%)	$\mathcal{R}_o$(%)	$\mathcal{F}_o$(%)	$\mathcal{A}_o$(%)
LSTM	82.41	81.78	82.02	83.10
BiLSTM	84.40	83.93	84.14	84.64
CNN-Arch-III	94.00	94.30	94.10	93.90

achieved relatively better classification results than others, as shown in Tables 14 and 15. Also, the proposed model has the scalability to classify large industrial program codes. Typically, industrial program codes are quite long and contain many functions and classes. As these functions may contain different algorithms, the proposed model can be useful for classifying codes at the function-level. It can be seen that the proposed code classification model can be useful and scalable for various programming-related tasks.

5.3 Model usage in programming learning

One of our research objectives is how the model can help programmers learn to program in real-world environments. From this viewpoint, the proposed model has been developed. The experimental results indicate that the present study can be useful for programming learning. A considerable amount of programming code is regularly generated from various sources such as academia, industry, programming platforms, and the OJ. However, programmers often find it challenging to identify the algorithms in the reference program codes while learning and searching from a large number of codes. Therefore, knowing the program code algorithm can help programmers better understand the code and accelerate their learning progress. Here, the proposed code classification model can effectively assist programmers in identifying algorithms contained in program codes. Moreover, the proposed model can be integrated with various real-world programming learning platforms, including OJ systems.

5.4 Model usage in software engineering

Repositories of real-world program codes play a key role in building effective ML models in SE. ML models are suitable in various fields of SE, such as strategic decision making, rapid prototyping, design and analysis, bug detection, code review, bug fixing, code reuse, and intelligent programming assistants (IPA). In addition, ML-enabled IPA systems can provide the best relevant code examples, best practices, and related texts as just-in-time support. As a result, the importance of ML models in software development and their application in SE is increasing significantly [14, 75]. The proposed CNN model classifies program codes by identifying the algorithms contained in the codes. Therefore, this model can also be used directly/indirectly for various SE tasks such as code review, bug detection, code examples, and code refactoring. In particular, the proposed model can be used as a supporting component of other ML models in SE that deal with SFs of program codes.

5.5 Threats to validity

This study applied several novel ideas from data preprocessing to model development. The model achieved significant results in classifying program codes during the experiment. However, the proposed model may suffer due to the following reasons/threats: (i) variation in the list of feature tokens for other programming languages; (ii) different strategies for data preprocessing; (iii) different sets of programming problems; (iv) problem sets with other programming languages such as C, Python, Java, and C#; and (v) different values of hyperparameters and architectures of the CNN model.

In the follow-up work, we plan to validate the model's performance by addressing the threats above mentioned.

6 Conclusion and future work

We developed CNN models to classify the program codes based on the identified algorithms. Real-world program codes were collected from the AOJ system and utilized in all experimental tasks. The SFs of the program codes were extracted to learn the CNN models. They were converted to OBM, followed by several processing steps. Different sets of hyperparameters such as $\mathcal{CL}$, $\mathcal{LR}$, $\mathcal{AF}$, $\mathcal{BS}$ were used in CNN models in different combinations. The top-3 CNN models and their hyperparameters were selected based on the superior experimental results. In addition, a 10-fold cross-validation was performed to select the most suitable (topmost) CNN model and hyperparameters for further experiments. Subsequently, all the experiments with the best CNN model were performed on both datasets (A and B). The model achieved significant classification results for both datasets, an average $\mathcal{P}_o$, $\mathcal{R}_o$, and $\mathcal{F}_o$ score of 94.30%, 94.80%, and 94.50%, respectively for Dataset A, and an average $\mathcal{P}_o$, $\mathcal{R}_o$, and $\mathcal{F}_o$ score of 97.00%, 96.90%, and 96.90%, respectively, for Dataset B. Furthermore, the performance of the proposed CNN model was compared with those of other baseline models. Results indicate that the proposed model outperforms the referenced models. The results show that the proposed model is more scalable in classifying program codes of diverse algorithms. In addition, the model can be useful in classifying program codes of other procedural programming languages, such as C, Java, Python, and C#.

In the future, the code block sequence of program codes can be considered, instead of SFs, to investigate the model performance. Moreover, a multi-label classification model can be considered to classify program codes with multiple labels. In addition, the model can be used to evaluate large-scale industrial program codes.

Funding This research work was supported by the Japan Society for the Promotion of Science (JSPS) KAKENHI (Grant Number 19K12252).

Code Availability In this paper, all real-world program codes for the experimental tasks are collected from the AOJ platform. All codes can be accessed via the following reference URLs: https://onlinejudge.u-aizu.ac.jp, https://judge.u-aizu.ac.jp/onlinejudge/, and http://developers.u-aizu.ac.jp/index.

Declarations

Conflict of Interests The authors declare that they have no conflicts of interest.

References

1. Rahman MM, Watanobe Y, Kiran RU, Thang TC, Paik I (2021) Impact of practical skills on academic performance: a data-driven analysis. IEEE Access 9:139975–139993. https://doi.org/10.1109/ACCESS.2021.3119145
2. Medeiros RP, Ramalho GL, Falcão TP (2019) A systematic literature review on teaching and learning introductory programming in higher education. IEEE Trans Educ 62(2):77–90. https://doi.org/10.1109/TE.2018.2864133
3. Perera P, Tennakoon G, Ahangama S, Panditharathna R, Chathuranga B (2021) A systematic mapping of introductory programming languages for novice learners. IEEE Access 9:88121–88136. https://doi.org/10.1109/ACCESS.2021.3089560
4. Mehmood E, Abid A, Farooq MS, Nawaz NA (2020) Curriculum, teaching and learning, and assessments for introductory programming course. IEEE Access 8:125961–125981. https://doi.org/10.1109/ACCESS.2020.3008321
5. Watanobe Y, Rahman MM, Matsumoto T, Rage UK, Ravikumar P (2022) Online judge system: requirements, architecture, and experiences. Int J Softw Eng Knowl Eng 32(06):917–946. https://doi.org/10.1142/S0218194022500346
6. Trisovic A, Lau MK, Pasquier T, Crosas M (2022) A large-scale study on research code quality and execution. Sci Data 9(1):. https://doi.org/10.1038/s41597-022-01143-6
7. Teshima Y, Watanobe Y (2018) Bug detection based on lstm networks and solution codes. In: 2018 IEEE international conference on systems, man, and cybernetics (SMC), pp 3541–3546. https://doi.org/10.1109/SMC.2018.00599
8. Rahman MM, Watanobe Y, Nakamura K (2020) Source code assessment and classification based on estimated error probability using attentive lstm language model and its application in programming education. Appl Sci 10(8):2973. https://doi.org/10.3390/app10082973
9. Intisar CM, Watanobe Y (2018) Classification of online judge programmers based on rule extraction from self organizing feature map. In: 2018 9th international conference on awareness science and technology (iCAST), pp 313–318. https://doi.org/10.1109/ICAwST.2018.8517222
10. Intisar CM, Watanobe Y (2018) Cluster analysis to estimate the difficulty of programming problems. In: Proceedings of the 3rd international conference on applications in information technology. ICAIT'2018, pp 23–28. https://doi.org/10.1145/3274856.3274862
11. Rahman MM, Watanobe Y, Rage UK, Nakamura K (2021) A novel rule-based online judge recommender system to promote computer programming education. In: Fujita H, Selamat A, Lin JC-W, Ali M (eds) Advances and trends in artificial intelligence. From theory to practice, Springer, pp 15–27. https://doi.org/10.1007/978-3-030-79463-7_2
12. Saito T, Watanobe Y (2020) Learning path recommendation system for programming education based on neural networks. Int J Dis Educ Technol (IJDET) 18(1):36–64. https://doi.org/10.4018/IJDET.2020010103
13. Taibi F (2013) Reusability of open-source program code: a conceptual model and empirical investigation. SIGSOFT Softw. Eng. Notes 38(4):1–5. https://doi.org/10.1145/2492248.2492276
14. Wan Z, Xia X, Lo D, Murphy GC (2021) How does machine learning change software development practices? IEEE Trans Softw Eng 47(9):1857–1871. https://doi.org/10.1109/TSE.2019.2937083
15. Amershi S, Begel A, Bird C, DeLine R, Gall H, Kamar E, Nagappan N, Nushi B, Zimmermann T (2019) Software engineering for machine learning: a case study. In: 2019 IEEE/ACM 41st international conference on software engineering: software engineering in practice (ICSE-SEIP), pp 291–300. https://doi.org/10.1109/ICSE-SEIP.2019.00042
16. Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735
17. Schuster M, Paliwal K. K (1997) Bidirectional recurrent neural networks. IEEE Trans Signal Process 45(11):2673–2681. https://doi.org/10.1109/78.650093
18. Krizhevsky A, Sutskever I, Hinton G. E (2017) Imagenet classification with deep convolutional neural networks. Commun ACM 60(6):84–90. https://doi.org/10.1145/3065386
19. Gao H, Xiao J, Yin Y, Liu T, Shi J (2022) A mutually supervised graph attention network for few-shot segmentation: the perspective of fully utilizing limited samples. IEEE Trans Neural Netw Learn Syst :1–13
20. Gao H, Xu K, Cao M, Xiao J, Xu Q, Yin Y (2022) The deep features and attention mechanism-based method to dish healthcare under social iot systems: an empirical study with a hand-deep local–global net. IEEE Trans Comput Soc Syst 9(1):336–347
21. Xiao J, Xu H, Gao H, Bian M, Li Y (2021) A weakly supervised semantic segmentation network by aggregating seed cues: the multi-object proposal generation perspective. ACM Trans Multimed Comput Commun Appl 17(1s):1–19
22. Rahim MA, Islam MR, Shin J (2019) Non-touch sign word recognition based on dynamic hand gesture using hybrid segmentation and cnn feature fusion. Appl Sci 9(18):3790. https://doi.org/10.3390/app9183790
23. Ran X, Shan Z, Fang Y, Lin C (2019) An lstm-based method with attention mechanism for travel time prediction. Sensors 19(4):861. https://doi.org/10.3390/s19040861
24. Gao H, Qiu B, Duran Barroso RJ, Hussain W, Xu Y, Wang X (2022) Tsmae: a novel anomaly detection approach for internet of things time series data using memory-augmented autoencoder. In: IEEE Transactions on Network Science and Engineering, pp 1–1

25. Zhao H, Sun S, Jin B (2018) Sequential fault diagnosis based on lstm neural network. IEEE Access 6:12929–12939. https://doi.org/10.1109/ACCESS.2018.2794765
26. Gao H, Huang W, Liu T, Yin Y, Li Y (2022) Ppo2: location privacy-oriented task offloading to edge computing using reinforcement learning for intelligent autonomous transport systems. In: IEEE Transactions on Intelligent Transportation Systems, pp 1–14
27. Rahman MM, Kawabayashi S, Watanobe Y (2021) Categorization of frequent errors in solution codes created by novice programmers. SHS Web Conf 102:04014. https://doi.org/10.1051/shsconf/202110204014
28. Fan G, Diao X, Yu H, Yang K, Chen L, Vitiello A (2019) Software defect prediction via attention-based recurrent neural network. Sci Program 2019:14. https://doi.org/10.1155/2019/6230953
29. Terada K, Watanobe Y (2021) Code completion for programming education based on deep learning. Int J Comput Intell Stud 10(2-3):78–98. https://doi.org/10.1504/IJCISTUDIES.2021.115424
30. Ohashi H, Watanobe Y (2019) Convolutional neural network for classification of source codes. In: 2019 IEEE 13th international symposium on embedded multicore/many-core systems-on-chip (MCSoC), pp 194–200. https://doi.org/10.1109/MCSoC.2019.00035
31. Rahman MM, Watanobe Y, Nakamura K (2020) A neural network based intelligent support model for program code completion. Sci Program 2020:18. https://doi.org/10.1155/2020/7426461
32. Rahman MM, Watanobe Y, Nakamura K (2021) A bidirectional lstm language model for code evaluation and repair. Symmetry 13(2):247. https://doi.org/10.3390/sym13020247
33. Rahman MM, Watanobe Y, Nakamura K (2020) Evaluation of source codes using bidirectional lstm neural network. In: 2020 3rd IEEE international conference on knowledge innovation and invention (ICKII), pp 140–143. https://doi.org/10.1109/ICKII50300.2020.9318916
34. Yera R, Martínez L (2017) A recommendation approach for programming online judges supported by data preprocessing techniques. Appl Intell 47(2):277–290. https://doi.org/10.1007/s10489-016-0892-x
35. Wasik S, Antczak M, Badura J, Laskowski A, Sternal T (2018) A survey on online judge systems and their applications. ACM Comput Surv 51(1):1–34. https://doi.org/10.1145/3143560
36. Project CodeNet (2021) [Online] Available: https://github.com/IBM/Project_CodeNet. Accessed 10 Jan 2022
37. Li Y, Choi D, Chung J, Kushman N, Schrittwieser J, Leblond R, Eccles T, Keeling J, Gimeno F, Lago AD, Hubert T, Choy P, d'Autume CdM, Babuschkin I, Chen X, Huang P-S, Welbl J, Gowal S, Cherepanov A, Molloy J, Mankowitz D. J, Robson ES, Kohli P, de Freitas N, Kavukcuoglu K, Vinyals O (2022) Competition-Level Code Generation with AlphaCode. https://doi.org/10.48550/ARXIV.2203.07814
38. Lu S, Guo D, Ren S, Huang J, Svyatkovskiy A, Blanco A, Clement C, Drain D, Jiang D, Tang D, Li G, Zhou L, Shou L, Zhou L, Tufano M, Gong M, Zhou M, Duan N, Sundaresan N, Deng SK, Fu S, Liu S (2021) CodeXGLUE: a machine learning benchmark dataset for code understanding and generation. https://doi.org/10.48550/ARXIV.2102.04664
39. Chen T-L, Hsiao T-C, Kang T-C, Wu T-Y, Chen C-C (2020) Learning programming language in higher education for sustainable development: point-earning bidding method. Sustainability 12(11):4489. https://doi.org/10.3390/su12114489
40. Rahman MM, Watanobe Y, Matsumoto T, Kiran RU, Nakamura K (2022) Educational data mining to support programming learning using problem-solving data. IEEE Access 10:26186–26202. https://doi.org/10.1109/ACCESS.2022.3157288
41. Sun Q, Wu J, Liu K (2019) How are students' programming skills developed: an empirical study in an object-oriented course. In: Proceedings of the ACM turing celebration conference - China. ACM TURC '19. https://doi.org/10.1145/3321408.3322858
42. Qian Y, Lehman J (2017) Students' misconceptions and other difficulties in introductory programming: a literature review. ACM Trans Comput Educ 18(1):1–24. https://doi.org/10.1145/3077618
43. Xia BS (2017) A pedagogical review of programming education research: what have we learned. Int J Online Pedagog Course Des 7(1):33–42. https://doi.org/10.4018/IJOPCD.2017010103
44. Jordan MI, Mitchell TM (2015) Machine learning: trends, perspectives, and prospects. Science 349(6245):255–260. https://doi.org/10.1126/science.aaa8415
45. Salvaris M, Dean D, Tok WH (2018) Microsoft AI platform. Apress, pp 79–98. https://doi.org/10.1007/978-1-4842-3679-6_4
46. Schelter S, Biessmann F, Januschowski T, Salinas D, Seufert S, Szarvas G (2018) On challenges in machine learning model management. IEEE Data Eng Bull 41:5–15
47. Martin Z (2019) Rules of machine learning: best practices for ML engineering https://developers.google.com/machine-learning/guides/rules-of-ml/. Accessed 25 Dec 2021
48. Ma L, Juefei-Xu F, Zhang F, Sun J, Xue M, Li B, Chen C, Su T, Li L, Liu Y, Zhao J, Wang Y (2018) DeepGauge: multi-granularity testing criteria for deep learning systems. Association for Computing Machinery, New York, pp 120–131. https://doi.org/10.1145/3238147.3238202
49. Pei K, Cao Y, Yang J, Jana S (2019) Deepxplore: automated whitebox testing of deep learning systems. Commun ACM 62(11):137–145. https://doi.org/10.1145/336.1566
50. Xie X, Ho JWK, Murphy C, Kaiser G, Xu B, Chen TY (2011) Testing and validating machine learning classifiers by metamorphic testing. J Syst Softw 84(4):544–558. https://doi.org/10.1016/j.jss.2010.11.920
51. Ma S, Liu Y, Lee W-C, Zhang X, Grama A (2018) Mode: Automated neural network model debugging via state differential analysis and input selection. In: Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering. ESEC/FSE 2018, Association for Computing Machinery, pp 175–186. https://doi.org/10.1145/3236024.3236082
52. Mou L, Li G, Zhang L, Wang T, Jin Z (2016) Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the Thirtieth AAAI conference on artificial intelligence. AAAI'16, pp 1287–1293. https://doi.org/10.5555/3015812.3016002
53. Wan Y, Shu J, Sui Y, Xu G, Zhao Z, Wu J, Yu P (2019) Multi-modal attention network learning for semantic source code retrieval. In: 2019 34th IEEE/ACM international conference on automated software engineering (ASE), pp 13–25. https://doi.org/10.1109/ASE.2019.00012
54. Hindle A, Barr ET, Su Z, Gabel M, Devanbu P (2012) On the naturalness of software. In: Proceedings of the 34th international conference on software engineering. ICSE '12, pp 837–847
55. Raychev V, Vechev M, Yahav E (2014) Code completion with statistical language models. ACM SIGPLAN Notices 49(6):419–428. https://doi.org/10.1145/2666356.2594321
56. Bui N, Jiang L, Yu Y Cross-language learning for program classification using bilateral tree-based convolutional neural networks. https://www.aaai.org/ocs/index.php/WS/AAAIW18/paper/view/17338/15660
57. Lu M, Wang Y, Tan D, Zhao L (2021) Student program classification using gated graph attention neural network. IEEE Access 9:87857–87868. https://doi.org/10.1109/ACCESS.2021.3063475
58. Ugurel S, Krovetz R, Giles CL (2002) 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. KDD '02, ACM, pp 632–638. https://doi.org/10.1145/775047.775141
59. Tian K, Revelle M, Poshyvanyk D (2009) Using latent dirichlet allocation for automatic categorization of software. In: 2009 6th IEEE international working conference on mining software repositories, pp 163–166. https://doi.org/10.1109/MSR.2009.5069496
60. Alreshedy K, Dharmaretnam D, German DM, Srinivasan V, Gulliver TA (2018) Scc: automatic classification of code snippets. In: 2018 IEEE 18th international working conference on source code analysis and manipulation (SCAM), pp 203–208. https://doi.org/10.1109/SCAM.2018.00031
61. Reyes J, Ramírez D, Paciello J (2016) Automatic classification of source code archives by programming language: a deep learning approach. In: 2016 international conference on computational science and computational intelligence (CSCI), pp 514–519. https://doi.org/10.1109/CSCI.2016.0103
62. Gilda S (2017) Source code classification using neural networks. In: 2017 14th international joint conference on computer science and software engineering (JCSSE), pp 1–6. https://doi.org/10.1109/JCSSE.2017.8025917
63. Shalaby M, Mehrez T, El Mougy A, Abdulnasser K, Al-Safty A (2017) Automatic algorithm recognition of source-code using machine learning. In: 2017 16th IEEE international conference on machine learning and applications (ICMLA), pp 170–177. https://doi.org/10.1109/ICMLA.2017.00033
64. Taherkhani A (2010) Recognizing sorting algorithms with the c4.5 decision tree classifier. In: 2010 IEEE 18th International Conference on Program Comprehension, pp 72–75. https://doi.org/10.1109/ICPC.2010.11
65. LeClair A, Eberhart Z, McMillan C (2018) Adapting neural text classification for improved software categorization. In: 2018 IEEE international conference on software maintenance and evolution (ICSME), pp 461–472. https://doi.org/10.1109/ICSME.2018.00056
66. Xu A, Dai T, Chen H, Ming Z, Li W (2018) Vulnerability detection for source code using contextual lstm. In: 2018 5th international conference on systems and informatics (ICSAI), pp 1225–1230. https://doi.org/10.1109/ICSAI.2018.8599360
67. Kim Y (2014) Convolutional neural networks for sentence classification. In: Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), Association for Computational Linguistics, pp 1746–1751. https://doi.org/10.3115/v1/D14-1181
68. Dey S, Singh AK, Prasad DK, Mcdonald-Maier KD (2019) Socodecnn: program source code for visual cnn classification using computer vision methodology. IEEE Access 7:157158–157172. https://doi.org/10.1109/ACCESS.2019.2949483
69. Watanobe Y (2018) Aizu online judge available: https://onlinejudge.u-aizu.ac.jp/. Accessed 1 Feb 2022
70. Aizu Online Judge (2004) Developers site (API) Available: http://developers.u-aizu.ac.jp/index. Accessed 1 Feb 2022
71. Puri R, Kung DS, Janssen G, Zhang W, Domeniconi G, Zolotov V, Dolby J, Chen J, Choudhury MR, Decker L, Thost V, Buratti L, Pujar S, Finkler U (2021) Project codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. arXiv: 2105.12655
72. Chevtchenko SF, Vale RF, Macario V, Cordeiro FR (2018) A convolutional neural network with feature fusion for real-time hand posture recognition. Appl Soft Comput 73:748–766. https://doi.org/10.1016/j.asoc.2018.09.010
73. Lu L, Shin Y, Su Y, Em Karniadakis G (2020) Dying relu and initialization: theory and numerical examples. Commun Comput 28(5):1671–1706. https://doi.org/10.4208/cicp.OA-2020-0165
74. Rahman MM, Watanobe Y, Kiran RU, Kabir R (2021) A stacked bidirectional lstm model for classifying source codes built in mpls. In: Machine learning and principles and practice of knowledge discovery in databases, pp 75–89. https://doi.org/10.1007/978-3-030-93733-1_5
75. Borges O, Couto J, Ruiz D, Prikladnicki R (2020) How machine learning has been applied in software engineering? In: Proceedings of the 22nd international conference on enterprise information systems - volume 2: ICEIS, pp 306–313. https://doi.org/10.5220/0009417703060313

Yutaka Watanobe is currently a Senior Associate Professor at the School of Computer Science and Engineering, The University of Aizu, Japan. He received his M.S. and Ph.D. degrees from The University of Aizu in 2004 and 2007 respectively. He was a Research Fellow of the Japan Society for the Promotion of Science (JSPS) at The University of Aizu in 2007. He is now a director of i-SOMET. He was a coach of four ICPC World Final teams. He is a developer of the Aizu Online Judge (AOJ) system. His research interests include intelligent software, programming environment, smart learning, machine learning, data mining, cloud robotics, and visual languages. He is a member of IEEE, IPSJ.

Md. Mostafizer Rahman is currently pursuing his Ph.D. degree in the Department of Computer and Information Systems, the University of Aizu, Aizu-Wakamatsu City, Fukushima, Japan. He is also working (on study leave) at Dhaka University of Engineering & Technology, Gazipur, Bangladesh. He received his B.Sc. and M.Sc. engineering degrees in the Computer Science and Engineering Department from Hajee Mohammad Danesh Science & Technology University, Dinajpur, Bangladesh, and Dhaka University of Engineering & Technology, Gazipur, Bangladesh, in 2009 and 2014, respectively. His research interests include machine learning, machine learning application in programming, natural language processing, data mining, and big data analytics.

Md. Faizul Ibne Amin received his B.Sc. degree in engineering in the Department of Computer Science and Engineering, East-West University, Dhaka, Bangladesh, in 2018, and his M.Sc. degree in Graduate School of Computer Science and Engineering from the Department of Computer and Information Systems, Aizu-Wakamatsu, Fukushima, Japan, in 2022. He is currently enrolled in the Ph.D. degree in the Database Systems Laboratory, Department of Computer and Information Systems, the University of Aizu, Aizu-Wakamatsu, Fukushima, Japan. His research interests include machine learning, deep learning, blockchain, payment channel network, educational data mining, and machine learning application in programming.

Mr. Raihan Kabir received his B.Sc. degree in computer science and engineering from the University of Asia Pacific (UAP), Dhaka, Bangladesh, in 2019, and his M.Sc. degree in Computer and Information Systems from the University of Aizu, Fukushima, Japan, in 2021. Currently, he is pursuing his Ph.D. degree in computer science and engineering at the University of Aizu, Fukushima, Japan. His research areas include Machine learning, Image processing, Cloud robotics, and Robotic navigation. He also participated in World Robot Sumit (WRS) 2020 robotic contest, and his team REL-UoA-JAEA secured 3rd position in the Disaster Robotics Category.

Affiliations

Yutaka Watanobe[1] · **Md. Mostafizer Rahman**[2,3] · **Md. Faizul Ibne Amin**[2] · **Raihan Kabir**[2]

Yutaka Watanobe
yutaka@u-aizu.ac.jp

Md. Faizul Ibne Amin
aminfaizul007@gmail.com

Raihan Kabir
raihan.kabir.cse@gmail.com

1 Department of Computer Science and Engineering, The University of Aizu, Aizu-Wakamatsu City, 965-8580, Fukushima, Japan

2 Graduate Department of Computer Science and Engineering, The University of Aizu, Aizu-Wakamatsu City, 965-8580, Fukushima, Japan

3 Department of Computer Science and Engineering, Dhaka University of Engineering & Technology, Gazipur, 1707, Bangladesh