Multilingual Program Code Classification Using n-Layered Bi-LSTM Model With Optimized Hyperparameters

Md. Mostafizer Rahman and Yutaka Watanobe, *Member, IEEE*

***Abstract*—Programmers are allowed to solve problems using multiple programming languages, resulting in the accumulation of a huge number of multilingual solution codes. Consequently, identifying codes from this vast archive of multilingual codes is a challenging and non-trivial task. Considering the codes' complexity compared to natural languages, conventional language models have had limited success. Deep neural network models have achieved state-of-the-art performance in programming-related tasks. However, the multilingual code classification based on the problem name or algorithm remains an open problem. This paper presents a novel multilingual program code classification model for the code classification task based on algorithms and problem names. First, a layered bidirectional long short-term memory model is designed to better understand the complex code context. Second, preprocessing, tokenization, and encoding processes are performed on real-life datasets. Next, clean and trainable formatted data are prepared. Finally, experiments are conducted on real-life datasets (e.g., sorting, searching, graphs and trees, numerical computations, basic data structures, and their combinations) with optimized hyperparameter settings. The results show that the proposed model can effectively improve the code classification accuracy compared to other baseline models.**

***Index Terms*—Multilingual program code, code classification, bidirectional lstm (bi-lstm), layered bi-lstm, programming learning.**

I. Introduction

PROGRAMMING is one of the key techniques for developing the modern information technology. Millions of codes are regularly generated in industrial and academic institutions [1], [2]. Programmers solve a single programming problem using different algorithms and programming languages and consider the instructions and constraints of the problem when writing the code. As a result, diverse and multilingual source codes[1] are regularly accumulated and pushed out to the cloud repository [3]. The manual classification of the huge number of diverse multilingual source code is a challenging and non-trivial task. Although many engineering models and approaches have been proposed to guide development, code writing is often expensive and error-prone [4]. Thus, it is important to develop a classification model that can better recognize the features and context of diverse codes to assist programmers. In particular, from a programming education perspective, the code classification model can provide students, teachers, and instructors with additional benefits in finding or recognizing relevant codes based on algorithms/problem names from large code repositories to accelerate programming learning. In software engineering (SE), in turn, the code classification model can help find appropriate modules to speed up software development. The model can be used at the functional-level in the software development phases.

A considerable number of industrial and academic studies on programming-related tasks to assist and alleviate the various programming challenges faced by programmers, especially coding. Some examples are: automatic localization of errors in solution codes [5], [6], [7], [8], editing changes in solution codes [9], [10], [11], [12], code refactoring [13], [14], [15], mathematics-based formal methods and techniques for generating solution code according to the code specification [16], [17], [18], code completion [19], identification of errors (e.g.., logical and syntactic) in solution codes [20], code evaluation and repair [21], and classification of codes based on errors, algorithms, languages, domains (e.g., network, game, word, and science), and code snippets [3], [22], [23], [24], [25], [26]. Classification methods are typically divided into two families: supervised learning (SL) and unsupervised learning (UL) [27]. In SL, the models are trained with known pair of tuples (e.g., $< \textit{input, label} >$). Let $\chi = \{\chi_1, \chi_2, \chi_3, \ldots, \chi_n\}$ be the set of solution codes and $\tau = \{\tau_1, \tau_2, \tau_3, \ldots, \tau_n\}$ be the set of the corresponding labels (problem name/algorithm) of set χ. Based on this definition, the output function of SL is written as $\tau = \Phi(\chi)$, where Φ is a mapping function and the τ output depends on the χ input. In contrast, UL models are trained without known corresponding input data labels, and the data are processed by mathematical methods based on their similarity features [2]. In practice, collecting sufficient labeled data for a given task is often expensive and time-consuming. In order to obtain meaningful results with SL, sufficient and high-quality data are required for model training; otherwise, low-quality data may adversely

Manuscript received 2 June 2023; revised 3 October 2023; accepted 14 October 2023. This work was supported by the Japan Society for the Promotion of Science (JSPS) KAKENHI under Grant 23H03508. *(Corresponding author: Md. Mostafizer Rahman.)*

Md. Mostafizer Rahman is with the Department of Computer and Information Systems, The University of Aizu, Fukushima 965-0006, Japan, and also with Information and Communication Technology Cell, Dhaka University of Engineering & Technology, Gazipur 1707, Bangladesh (e-mail: mostafiz26@gmail.com).

Yutaka Watanobe is with the Department of Computer and Information Systems, The University of Aizu, Fukushima 965-0006, Japan (e-mail: yutaka@u-aizu.ac.jp).

Digital Object Identifier 10.1109/TETCI.2023.3336920

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

affect its effectiveness. This problem is especially important in classification tasks with complex data and often referred to as the "*cold start problem*" [27].

In recent years, deep neural network (DNN) models have achieved profound success in various tasks, including computer vision [28], image classification with a bidirectional long short-term memory (Bi-LSTM) network [29], [30], high-dimensional anomaly detection [31], [32], healthcare services using the data of Internet-of-Things systems [33], and online learning algorithms with a long short-term memory (LSTM) network [34]. Furthermore, many approaches have been proposed to better understand the program code [35]. Watanobe et al. [36] proposed a source code classification model using convolutional neural networks. The source codes in the C++ programming language were used in their experiments. Bi-LSTM and LSTM language models were also employed for the source code classification tasks [19], [20], [21]. Stochastic language models (SLMs) have achieved widespread success in natural language processing (NLP), speech recognition, language translation, and handwriting recognition [19]. The performance of SLMs (e.g., n-gram, bi-gram, $skip$-gram, and glove [37], [38]) heavily depends on a rich text corpus. However, considering the program code complexity, the different code structures using different programming languages, and code corpus limitation, SLMs have not achieved the same significant results as like NLP tasks.

As a remedy, recurrent neural networks (RNNs) have been introduced. RNNs have a built-in network memory that can store past information [39], but they cannot process long-term dependent information because the gradient exponentially increases or decreases during training. This problem is referred to as the "*gradient vanishing and exploding*" problem [40], [41]. It prevents the capturing of long-term dependent information and significantly degrades the RNN performance in real-world implementations [42]. To address the problem, the LSTM network has been introduced. It has a novel network architecture with four control gates (i.e., input, output, forget, and cell state) that can be used to overcome the "*gradient vanishing and exploding*" problem [42], [43]. Despite the good performance of the LSTM, it only processes the information in one direction from past to future, which means that the LSTM is unidirectional [44]. Resolving this problem required the introduction of Bi-LSTM network [45] that processes information in both forward and backward directions. In Bi-LSTM, two independent hidden layers (i.e., forward and backward) are connected to the same input. The results of these two layers are concatenated for the output. In practice, Bi-LSTM models have shown a much better performance compared to LSTM [44].

In this paper, we propose an n-layered Bi-LSTM model for the program code classification tasks. In n-layered Bi-LSTM model, an n number of Bi-LSTM layers is used, where the output of the hidden state of each Bi-LSTM layer is given as the input to the next Bi-LSTM layer. The "*weight update*" formula is similar with that in the original Bi-LSTM. The deep layered architecture of the n-layered Bi-LSTM model allows the extraction of more complex features based on previous layers. This layered Bi-LSTM model mechanism understands the complex context and features of data. The solution codes contain functions, classes, keywords, tokens, characters, numbers, operators, and variables with long- and short-term dependencies; hence, the n-layered Bi-LSTM model structure can accurately capture the dependencies and the complex context of solution codes.

We validate the performance of the proposed n-layered Bi-LSTM model by creating several datasets (i.e., sorting, searching, graphs and trees, numerical computations, basic data structures and their combination) with real-life solution codes collected from an online judge system. The datasets are composed of multilingual (i.e., approximately 15 programming languages) and solution codes with various algorithms, which ensure the high diversity of the datasets. The experimental results suggest that the performance of the state-of-the-art models (i.e., LSTM and Bi-LSTM) on our dataset is significantly inferior to that of the n-layered Bi-LSTM model. Moreover, the network hyperparameters are fine-tuned during the model experiments. The evaluation results indicate that none of the compared state-of-the-art models outperform the n-layered Bi-LSTM model.

The main contributions of this study are as follows:

- We propose a novel n-layered Bi-LSTM model for the code classification task. We introduce herein a new input supply from one Bi-LSTM layer to another. The deep structure of the n-layered Bi-LSTM model can understand the complex context and features of the solution codes.
- The experimental results show a substantial improvement of the classification task on highly diverse solution codes compared to other state-of-the-art models (e.g., LSTM and Bi-LSTM).
- We create datasets using real-life solution codes that can be useful for other programming-related studies such as code generation, refactoring, code translation, and error detection.

The remainder of this paper is organized as follows: Section II describes the background and the theoretical foundations of the NLP and DNN models closely related to this study; Section III presents the problem statements and the motivation of our study; Section IV explains the details of our proposed program code classification approach. Section V provides the datasets, evaluation metrics, implementation details, results, and discussion; and Section VII concludes this study with a note on the future work.

II. Background and Theoretical Foundation

This section provides a brief introduction to the background and theoretical foundations of this study. Hence, the mathematical representations of the SLM, RNNs, LSTM, and Bi-LSTM models for the sequential language modeling tasks are presented.

A. n-Gram Language Model

The n-gram model is a popular language model in NLP tasks. It predicts the next words based on the word sequence probability. Let $a = \{a_1, a_2, a_3, \ldots, a_n\}$ be the set of words of sequence a, where a_i is a single word. The probability of the entire word sequence $\psi(a)$ is calculated as follows by the chain

rule of probability.

$$\psi(a_1^n) = \psi(a_1)\psi(a_2|a_1)\psi(a_3|a_1^2)\psi(a_4|a_1^3)\cdots\psi(a_n|a_a^{n-1}) = \prod_{j=1}^{n}(a_j|a_1^{j-1}) \quad (1)$$

If the probability of a word $\psi(a_i)$ depends on the preceding words $\psi(a_1^{i-1})$, it can be described through the Markov Assumption.

$$\psi(a_i|a_1^{i-1}) \approx \psi(a_i|a_{i-1}) \quad (2)$$

The n-gram uses (3) as the conditional probability of the next word in a sequence. In the following equation, N is used to indicate the n-gram size, such as $N = 2$ for bi-gram, $N = 3$ for tri-gram, and so on.

$$\prod_{j=1}^{n}(a_j|a_1^{j-1}) \approx \prod_{j=1}^{n}(a_j|a_{j-N+1}^{j-1}) \quad (3)$$

In practice, smoothing techniques are used to estimate the maximum likelihood using (4), where $\varphi(\cdot)$ computes the count of words.

$$\psi(a_j|a_{j-N+1}^{j-1}) = \frac{\varphi(a_{j-N+1}^{j-1}a_j)}{\varphi(a_{j-N+1}^{j-1})} \quad (4)$$

The language model's performance can be evaluated using the cross-entropy ($\mathcal{C}_e$) calculation, where a lower $\mathcal{C}_e$ value indicates a better language model, and vice versa [46].

$$\mathcal{C}_e \approx -\frac{1}{n}\sum_{j=1}^{n} log_2\psi(a_j|a_{j-N+1}^{j-1}) \quad (5)$$

B. Recurrent Neural Networks

An RNN is a neural network that can process dependent sequential data $x(t) = x(1), x(2), x(3), \ldots, x(T-1), x(T)$ by using its ingrained network "$memory$". This "$memory$" captures all previously calculated information. The basic RNN structure is described by the following set of equations [34]:

$$\boldsymbol{h}_t = d(\boldsymbol{U}^{(h)}\boldsymbol{x}_t + \boldsymbol{G}^{(h)}\boldsymbol{h}_{t-1}) \quad (6)$$

$$\hat{\boldsymbol{y}}_t = e(\boldsymbol{V}^{(y)}\boldsymbol{h}_t) \quad (7)$$

where $\boldsymbol{x}_t \in \mathbb{R}^j$ is the input vector; $\boldsymbol{h}_t \in \mathbb{R}^k$ is the hidden state vector; and $\hat{\boldsymbol{y}}_t \in \mathbb{R}^k$ is the output vector. Functions $d(.)$ and $e(.)$ are used for the non-linearity and output of the network. The widely used functions are $tanh(.)$ and $softmax(.)$. The coefficient weight matrices of $\boldsymbol{U}$, $\boldsymbol{G}$, and $\boldsymbol{V}$ are $\boldsymbol{U} \in \mathbb{R}^{j\times k}$, $\boldsymbol{G} \in \mathbb{R}^{k\times k}$, and $\boldsymbol{V} \in \mathbb{R}^{k\times k}$, respectively.

1) Gradient Vanishing and Exploding: Two processes are executed in a single time step: (i) forward, and (ii) backward passes. In the forward pass, the loss function ($\wp_t$), hidden layer state ($\boldsymbol{h}_t$), and output ($\hat{\boldsymbol{y}}_t$) are computed. The loss between the estimated ($\hat{\boldsymbol{y}}_t$) and true ($\boldsymbol{y}_t$) labels is calculated using $\wp_t$. The total loss is described as $\boldsymbol{L} = \sum_i \wp_i(\hat{\boldsymbol{y}}_t, \boldsymbol{y}_t)$. In the backward pass, the loss function gradient for each weight matrix ($\delta\boldsymbol{L}/\delta\boldsymbol{U}$, $\delta\boldsymbol{L}/\delta\boldsymbol{G}$, $\delta\boldsymbol{L}/\delta\boldsymbol{V}$) is computed to update weight matrices $\boldsymbol{U}$, $\boldsymbol{G}$, and $\boldsymbol{V}$ using the backpropagation algorithm (i.e., backpropagation through time). In the backward pass, the gradients are backpropagated through time and layers. All past contributions are summed up to the current contribution:

$$\frac{\delta\boldsymbol{L}}{\delta\boldsymbol{G}} = \sum_{i=0}^{T}\frac{\delta\wp_i}{\delta\boldsymbol{G}} = \sum_{i=0}^{T}\left(\prod_{i=m+1}^{p}\frac{\delta\boldsymbol{h}_i}{\delta\boldsymbol{h}_{i-1}}\right)\frac{\delta\boldsymbol{h}_m}{\delta\boldsymbol{G}} \quad (8)$$

where the contribution of a state (at time step m) to the gradient of the total loss $\boldsymbol{L}$ is calculated. Equation (8) has two erratic cases during backpropagation: (i) if $\left\|\frac{\delta\boldsymbol{h}_i}{\delta\boldsymbol{h}_{i-1}}\right\|_2 < 1$, the gradient vanishes or disappears; and (ii) if $\left\|\frac{\delta\boldsymbol{h}_i}{\delta\boldsymbol{h}_{i-1}}\right\|_2 > 1$, the gradient explodes.

C. Long Short-Term Memory Neural Networks

LSTM networks are specialized RNNs and used in various complex problem domains, such as speech recognition and machine translation. LSTMs can remember long dependent input sequences and overcome the gradient vanishing and explosion problems [42]. The LSTM architecture is described by the following set of equations [42]:

$$\breve{\boldsymbol{c}}_t = d(\boldsymbol{U}^{(c)}\boldsymbol{x}_t + \boldsymbol{G}^{(c)}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{(c)}) \quad (9)$$

$$\boldsymbol{f}_t = \sigma(\boldsymbol{U}^{(f)}\boldsymbol{x}_t + \boldsymbol{G}^{(f)}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{(f)}) \quad (10)$$

$$\boldsymbol{i}_t = \sigma(\boldsymbol{U}^{(i)}\boldsymbol{x}_t + \boldsymbol{G}^{(i)}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{(i)}) \quad (11)$$

$$\boldsymbol{c}_t = \boldsymbol{i}_t \odot \breve{\boldsymbol{c}}_t + \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} \quad (12)$$

$$\boldsymbol{o}_t = \sigma(\boldsymbol{U}^{(o)}\boldsymbol{x}_t + \boldsymbol{G}^{(o)}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{(o)}) \quad (13)$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot d(\boldsymbol{c}_t) \quad (14)$$

where $\boldsymbol{c}_t \in \mathbb{R}^k$ is the state vector; $\boldsymbol{x}_t \in \mathbb{R}^j$ is the input vector; and $\boldsymbol{h}_t \in \mathbb{R}^k$ is the output vector. Here, $\boldsymbol{c}_t$, $\boldsymbol{f}_t$, $\boldsymbol{i}_t$, and $\boldsymbol{o}_t$ are the cell state, forget, input, and output gates, respectively. $\breve{\boldsymbol{c}}_t \in \mathbb{R}^k$ is the candidate state obtained by the nonlinear function. Function $d(.)$ stands for the $tanh(.)$ function and is applied pointwise to the input vectors. Similarly, function $\sigma(.)$ stands for the sigmoid function and is applied pointwise to the vector elements. The coefficient weight matrices and vectors are as follows: $\boldsymbol{U}^{(c)} \in \mathbb{R}^{j\times k}$, $\boldsymbol{G}^{(c)} \in \mathbb{R}^{k\times k}$, $\boldsymbol{b}^{(c)} \in \mathbb{R}^k$, $\boldsymbol{U}^{(f)} \in \mathbb{R}^{j\times k}$, $\boldsymbol{G}^{(f)} \in \mathbb{R}^{k\times k}$, $\boldsymbol{b}^{(f)} \in \mathbb{R}^k$, $\boldsymbol{U}^{(i)} \in \mathbb{R}^{j\times k}$, $\boldsymbol{G}^{(i)} \in \mathbb{R}^{k\times k}$, $\boldsymbol{b}^{(i)} \in \mathbb{R}^k$, $\boldsymbol{U}^{(o)} \in \mathbb{R}^{j\times k}$, $\boldsymbol{G}^{(o)} \in \mathbb{R}^{k\times k}$, and $\boldsymbol{b}^{(o)} \in \mathbb{R}^k$.

D. Bidirectional Long Short-Term Memory Neural Networks

Bi-LSTM extends the unidirectional LSTM with a new hidden layer that passes information in the backward direction [45]. Thus, the forward hidden layer ($\overrightarrow{H}$) starts with the first token of the sequence, while the backward hidden layer ($\overleftarrow{H}$) starts with the last token. In other words, the connection flow from one hidden layer to another hidden layer is in a reverse temporal order.

The internal structure of the Bi-LSTM model is useful for understanding the complex context of dependent information, such as time series and language modeling tasks. Fig. 1 presents the basic architecture of the Bi-LSTM. The Bi-LSTM model

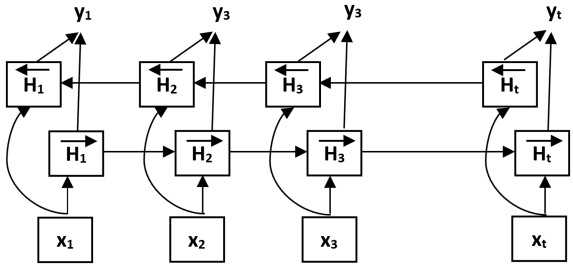


Fig. 1. Basic Bi-LSTM architecture.

architecture [45] is described by the following set of equations:

$$\overrightarrow{\boldsymbol{h}}_t = d(\boldsymbol{U}^{(fw)}\boldsymbol{x}_t + \boldsymbol{G}^{(fw)}\overrightarrow{\boldsymbol{h}}_{t-1} + \boldsymbol{b}^{(fw)}) \quad (15)$$

$$\overleftarrow{\boldsymbol{h}}_t = d(\boldsymbol{U}^{(bw)}\boldsymbol{x}_t + \boldsymbol{G}^{(bw)}\overleftarrow{\boldsymbol{h}}_{t+1} + \boldsymbol{b}^{(bw)}) \quad (16)$$

where the coefficient weight matrices and vectors are as follows: $\boldsymbol{U}^{(fw)} \in \mathbb{R}^{j\times k}$, $\boldsymbol{G}^{(fw)} \in \mathbb{R}^{k\times k}$, $\boldsymbol{b}^{(fw)} \in \mathbb{R}^{k}$, $\boldsymbol{U}^{(bw)} \in \mathbb{R}^{j\times k}$, $\boldsymbol{G}^{(bw)} \in \mathbb{R}^{k\times k}$, and $\boldsymbol{b}^{(bw)} \in \mathbb{R}^{k}$. Function $d(.)$ is an activation function usually set to the hyperbolic tangent function (i.e., $tanh(.)$). Also, the function $d(.)$ applies input vectors pointwise. Here, two hidden states ($\overrightarrow{\boldsymbol{h}}_t$ and $\overleftarrow{\boldsymbol{h}}_t$) are concatenated as $\boldsymbol{h}_t$ and fed to the output layer.

$$\boldsymbol{h}_t = \overrightarrow{\boldsymbol{h}}_t \oplus \overleftarrow{\boldsymbol{h}}_t \quad (17)$$

where $\boldsymbol{h}_t \in \mathbb{R}^{j\times 2k}$ is the weight matrix of the hidden state. Finally, the output $\boldsymbol{o}_t \in \mathbb{R}^{j\times q}$ is computed in the output layer (q: number of outputs) described as follows:

$$\boldsymbol{o}_t = e(\boldsymbol{h}_t\boldsymbol{V}^{(o)} + \boldsymbol{b}^{(o)}) \quad (18)$$

where the coefficient weight matrices and vectors are as follows: $\boldsymbol{V}^{(o)} \in \mathbb{R}^{2k\times q}$ and $\boldsymbol{b}^{(o)} \in \mathbb{R}^{q}$. Function $e(.)$ is the activation function of the output layer, which is typically set to the softmax function (i.e., $softmax()$).

III. Problem Statement and Motivation

In recent years, DNN models have achieved great success in NLP tasks [47], [48] due to the richness of the natural language corpus. The structure of natural languages is the same, which helps in the collection of a large corpus. In contrast, programming problems can be solved using many programming languages. Programmers write codes in their own style. There are no predefined ways to solve programming problems. Consequently, the heterogeneity of code structures and programming languages and the complexity of codes are not comparable to natural languages. Fig. 2 shows an example of a problem (selection sort) solved with three different programming languages (i.e., Java, C++, and Python). The variables, operators, methods, classes, keywords, input/output functions, header functions, and code composition of these solutions are completely different, making programming codes even more complex and diverse. For example, the number of heterogeneity/complexity features of a code can be described as follows: $\mathcal{H} = \sum_{l=1}^{L}(f + k + h + s + v + \cdots)$, where L is the number of programming languages; $f = i, i \in \mathbb{Z}, 1 \leq i \leq n$ is the number of functions; $k = j, j \in \mathbb{Z}, 1 \leq j \leq n$ is the keywords; $h = g, g \in \mathbb{Z}, 1 \leq g \leq n$ is the number of header functions; $s = l, l \in \mathbb{Z}, 1 \leq l \leq n$ is the number of structures; and $v = m, j \in \mathbb{Z}, 1 \leq m \leq n$ is the number of variables. Therefore, understanding these huge complex and diverse codes with DNN models is a difficult and non-trivial task compared to using natural languages. We are motivated to address the problem of "**how the DNN model can better understand the diverse and complex codes**" by developing a novel DNN model with a very deep structure. The deep structure helps to gain an in-depth understanding of code features through repeated input data (or codes) learning.

Algorithm 1: Overall Process of the Code Classification Approach.

Input: Solution Codes $\chi = \{\chi_1, \chi_2, \chi_3, \ldots, \chi_n\}$
Output: Classification of the solution code based on the class label
Initialize list $I[]$ for word tokenization, $\lambda = \{comments, whitespaces, tabs\}$
for each code $\mathfrak{s} \in \chi$ **do**
  Initialize list $\omega[]$ for words
  for each string/word $\vartheta \in \mathfrak{s}$ **do**
    if $\vartheta \in \lambda$ **then**
      Remove the word ϑ from code $\mathfrak{s}$
    else
      Assign words $\omega[] \longleftarrow \vartheta$
    end if
  end for
  for each word $\vartheta \in \omega$ **do**
  Assign token number for each word $I[] \longleftarrow \ell, \ell \in \mathbb{Z}, 1 \leq \ell \leq n$
  end for
end for
Embedding layer converts the sequence of token numbers $I[]$ of the codes into the Embedding matrix $\boldsymbol{E} \in \mathbb{R}^{v\times d}$ according to (19)
Vectorized code sequences ($\boldsymbol{E}$) are propagated into the n-layered Bi-LSTM and processed according to Algorithm 2.
High-dimensional vector representation $\boldsymbol{y}_t^{(n)}$ of the code sequences goes to the dense layer for further processing according to Fig. 4.
Predicts the class label of the solution code using Softmax at the output layer according to (41).

IV. Proposed Approach

The proposed code classification approach consists of two main phases, namely code preprocessing and classification, with the n-layered Bi-LSTM model. Algorithm 1 describes the overall code classification process as a pseudocode. The overview of the proposed method and the architecture of the n-layered Bi-LSTM model are described below.

```java
import java.util.*;

public class Main{
    Main(){
    Scanner sc = new Scanner(System.in);
    int n, t, s, cnt;
    int[] a;

    while(sc.hasNext()){
        n = sc.nextInt();
        a = new int[n];
        for(int i = 0; i < n; ++i)
        a[i] = sc.nextInt();

        cnt = 0;
        for(int i = 0; i < n; ++i){
        t = i;
        for(int j = i; j < n; ++j)
            if(a[j] < a[t]) t = j;

        if(a[t] != a[i]){
            s = a[i];
            a[i] = a[t];
            a[t] = s;
            ++cnt;
        }
        }
        for(int i = 0; i < n-1; ++i)
        System.out.printf("%d ", a[i]);
        System.out.println(a[n-1]);
        System.out.println(cnt);
    }
    }
    public static void main(String[] args){
    new Main();
    }
}
```

(a)

```cpp
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;

int n;
int a[100];
int main(){
    int count=0;
    int m;
    cin >> n;
    for(int i=0;i<n;i++){
        cin >> a[i];
    }
    for(int i=0;i<n;i++){
        m = i;
        for(int j=i;j<n;j++){
            if(a[m] > a[j]) m = j;
        }
        if(m != i){
            swap(a[i], a[m]);
            count++;
        }
    }
    cout << a[0];
    for(int i=1;i<n;i++){
        cout << " " << a[i];
    }
    cout << endl;
    cout << count << endl;
}
```

(b)

```python
import math

def swap(A, i, j):
    x = A[i]
    A[i] = A[j]
    A[j] = x

def sSort(A, N):
    cnt = 0
    for i in range(0, N-1):
        minj = i
        for j in range(i,N):
            if A[j] < A[minj]:
                minj = j
        if minj != i:
            swap(A,i,minj)
            cnt += 1
    return cnt

N = int(input().strip())
A = list(map(int,input().strip().split()))
cnt = sSort(A,N)
print(" ".join(map(str,A)))
print(cnt)
```

(c)

Fig. 2. Motivational example: A *selection sort* problem is solved with three different programming languages (Java, C++, and Python).

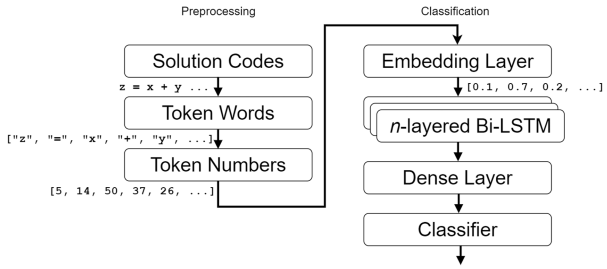


Fig. 3. Overview of the proposed code classification model.

A. Overview of the Classification Model

Fig. 3 presents an overview of the proposed model, where the left part shows the code preprocessing, and the right part is used for the model training and classification processes. The code preprocessing aims to remove irrelevant information from the original solution codes. It is considered as one of the most vital and primary tasks for the solution code classification. We obtained the ideal format of solution codes by following the code preprocessing steps of these studies [19], [20], [44]. To this end, we removed all irrelevant information (e.g., comments, tabs, spaces, and line breaks) from the solution codes for the model training and evaluation. Each solution code was then converted into a sequence of token words. Each token word was assigned a unique token number or integer index. Let $W = \{w_1, w_2, w_3, \ldots, w_v\}$ be the set of word sequences of a solution code and the corresponding mapping number be $O = \{o_1, o_2, o_3, \ldots, o_v\}$. To train the model, the lengths of the code sequences were kept the same by padding or truncation.

Meanwhile, the classification process starts with the embedding layer used to convert each integer index into a real-valued feature vector. These real-valued feature vectors of the code tokens are combined to form a matrix, called the embedding matrix [49]. Each row of the embedding representation indicates the original word of the code sequence. Note that the embedding representation in the matrix is identical for the same words. The embedding matrix is described by (19).

$$E_{v,d} = \begin{pmatrix} e_{1,1} & e_{1,2} & \cdots & e_{1,d} \\ e_{2,1} & e_{2,2} & \cdots & e_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ e_{v-1,1} & e_{v-1,2} & \cdots & e_{v-1,d} \\ e_{v,1} & e_{v,2} & \cdots & e_{v,d} \end{pmatrix} \tag{19}$$

where $\boldsymbol{E} \in \mathbb{R}^{v \times d}$ is the embedding matrix; v is the vocabulary size of the solution codes; and d is the embedding dimension of the dense vector. In this paper, the dimension $(v \times d)$ (i.e., $10,000 \times 200$) of the embedding matrix $\boldsymbol{E} \in \mathbb{R}^{v \times d}$ is used as the pretrained word embedding vector. Next, the vectorized code information (i.e., $< solution\ code, target\ label >$) is propagated to the DNN layers (i.e., n-layered Bi-LSTM and dense layer/fully connected layer) for feature learning of the solution codes. Finally, the output layer is used for the solution code classification.

B. Architecture of the N-Layered Bi-LSTM Neural Network

In this section, we present our proposed n-layered Bi-LSTM model architecture. A multilayer or stacking architecture of Bi-LSTM neural networks further improves the classification or regression performance [50], [51]. Moreover, existing works [52], [53], [54] have shown that deep hierarchical architectures with

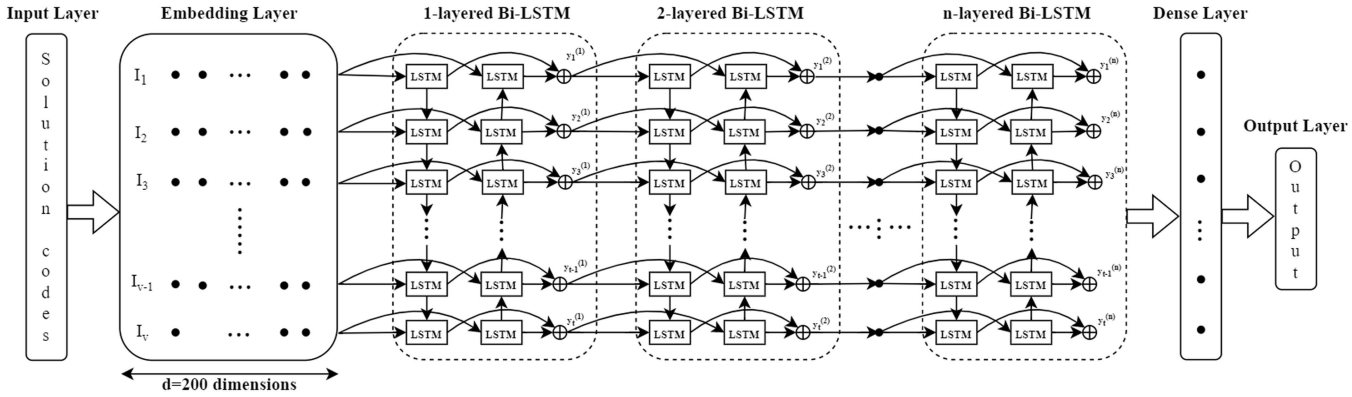


Fig. 4. Graphical architecture of the n-layered Bi-LSTM with input, embedding, dense, and output layers.

several hidden layers are more efficient than shallow ones because they can build higher-level representations of dependent sequential data. The layered Bi-LSTM architecture can provide a higher prediction performance by obtaining richer contextual information of the solution codes from both past and future sequences. The deep architectures consist of multiple hidden layers, where the output of one hidden layer is used as the input to the subsequent hidden layer. This layered mechanism enhances the neural network performance. We adopt this concept in this work. Fig. 4 illustrates the graphical architecture of the proposed n-layered Bi-LSTM for the code classification task.

In this scheme, the solution codes are first preprocessed in the input layer, where the lengths of the token sequences of each code are the same. Next, an embedding matrix ($\boldsymbol{E} \in \mathbb{R}^{v \times d}$) is created based on the code sequences in the embedding layer used as the input to the deep n-layered Bi-LSTM. In this architecture, for a particular time step t, the input is given to the hidden layers in the forward direction to learn information from the past. The same input data are fed to the hidden layers in the reverse direction to learn future information. The output y_t is obtained by concatenating the initial outputs of both the forward and reverse layers. The hidden units of the upper (next) layers take the output of the lower (previous) layers to determine the detailed contextual information from the solution codes. For example, $y_t^{(1)}$ is the output of the 1-layered Bi-LSTM used as the input for the 2-layered. Algorithm 2 describes the complete n-layered Bi-LSTM algorithm as a pseudocode. The n-layered Bi-LSTM is connected to the dense or fully-connected layer to further process the contextual representation of the solution codes. Finally, the output layer classifies the class label of the solution code using softmax.

Remark 1: In the n-layered Bi-LSTM architecture, the output of the lower layer becomes the input of the next layer.

For a given time step t, the output of the first layer of the n-layered Bi-LSTM architecture is defined by the following set of equations. First, the forward layer of 1-layered Bi-LSTM.

$$\boldsymbol{i}_t^{(1_f)} = \sigma(\boldsymbol{U}_i^{(1_f)}\boldsymbol{x}_t + \boldsymbol{G}_i^{(1_f)}\boldsymbol{h}_{t-1}^{(1)} + \boldsymbol{b}_i^{(1_f)}) \tag{20}$$

$$\boldsymbol{f}_t^{(1_f)} = \sigma(\boldsymbol{U}_f^{(1_f)}\boldsymbol{x}_t + \boldsymbol{G}_f^{(1_f)}\boldsymbol{h}_{t-1}^{(1)} + \boldsymbol{b}_f^{(1_f)}) \tag{21}$$

Algorithm 2: n-Layered Bidirectional LSTM (Bi-LSTM).

1: **Input:** Embedding vectors $\boldsymbol{E} \in \mathbb{R}^{v \times d}$ using tokenized code sequences $i \in I$ according to (19).
2: **Output:** $\boldsymbol{y}_t^{(n)}$: high-dimensional vector representation of code sequences, where t is the time step, and n is the number of layers in the Bi-LSTM.
3: **for** $n = 1 : N$ **do**
4: Calculate forward hidden state $\boldsymbol{h}_t^{(n_f)}$ of the n-layered Bi-LSTM according to (38)
5: Calculate backward hidden state $\boldsymbol{h}_t^{(n_b)}$ of the n-layered Bi-LSTM according to (39)
6: Concatenation of the both hidden states $\boldsymbol{y}_t^{(n)} = \boldsymbol{V}_y^{(n)}\boldsymbol{h}_t^{(n_f)} + \boldsymbol{V}_y^{(n)}\boldsymbol{h}_t^{(n_b)}$ of the n-layered Bi-LSTM according to (40)
7: **end for**
8: **Return** $\boldsymbol{y}_t^{(n)}$

$$\boldsymbol{o}_t^{(1_f)} = \sigma(\boldsymbol{U}_o^{(1_f)}\boldsymbol{x}_t + \boldsymbol{G}_o^{(1_f)}\boldsymbol{h}_{t-1}^{(1)} + \boldsymbol{b}_o^{(1_f)}) \tag{22}$$

$$\breve{\boldsymbol{c}}_t^{(1_f)} = d(\boldsymbol{U}_c^{(1_f)}\boldsymbol{x}_t + \boldsymbol{G}_c^{(1_f)}\boldsymbol{h}_{t-1}^{(1)} + \boldsymbol{b}_c^{(1_f)}) \tag{23}$$

$$\boldsymbol{c}_t^{(1_f)} = \boldsymbol{i}_t^{(1_f)} \odot \breve{\boldsymbol{c}}_t^{(1_f)} + \boldsymbol{f}_t^{(1_f)} \odot \boldsymbol{c}_{t-1}^{(1_f)} \tag{24}$$

$$\boldsymbol{h}_t^{(1_f)} = \boldsymbol{o}_t^{(1_f)} \odot d(\boldsymbol{c}_t^{(1_f)}) \tag{25}$$

where $\boldsymbol{U}_i^{(1_f)}$, $\boldsymbol{U}_f^{(1_f)}$, $\boldsymbol{U}_o^{(1_f)}$, and $\boldsymbol{U}_c^{(1_f)} \in \mathbb{R}^{j \times k}$ and $\boldsymbol{G}_i^{(1_f)}$, $\boldsymbol{G}_f^{(1_f)}$, $\boldsymbol{G}_o^{(1_f)}$, and $\boldsymbol{G}_c^{(1_f)} \in \mathbb{R}^{k \times k}$ are the coefficient weight matrices to learn. $\boldsymbol{b}_i^{(1_f)}$, $\boldsymbol{b}_f^{(1_f)}$, $\boldsymbol{b}_o^{(1_f)}$, and $\boldsymbol{b}_c^{(1_f)} \in \mathbb{R}^k$ are the bias vectors. Second, the backward layer of the 1-layered Bi-LSTM is described as follows:

$$\boldsymbol{i}_t^{(1_b)} = \sigma(\boldsymbol{U}_i^{(1_b)}\boldsymbol{x}_t + \boldsymbol{G}_i^{(1_b)}\boldsymbol{h}_{t+1}^{(1)} + \boldsymbol{b}_i^{(1_b)}) \tag{26}$$

$$\boldsymbol{f}_t^{(1_b)} = \sigma(\boldsymbol{U}_f^{(1_b)}\boldsymbol{x}_t + \boldsymbol{G}_f^{(1_b)}\boldsymbol{h}_{t+1}^{(1)} + \boldsymbol{b}_f^{(1_b)}) \tag{27}$$

$$\boldsymbol{o}_t^{(1_b)} = \sigma(\boldsymbol{U}_o^{(1_b)}\boldsymbol{x}_t + \boldsymbol{G}_o^{(1_b)}\boldsymbol{h}_{t+1}^{(1)} + \boldsymbol{b}_o^{(1_b)}) \tag{28}$$

$$\breve{\boldsymbol{c}}_t^{(1_b)} = d(\boldsymbol{U}_c^{(1_b)}\boldsymbol{x}_t + \boldsymbol{G}_c^{(1_b)}\boldsymbol{h}_{t+1}^{(1)} + \boldsymbol{b}_c^{(1_b)}) \tag{29}$$

$$\boldsymbol{c}_t^{(1_b)} = \boldsymbol{i}_t^{(1_b)} \odot \breve{\boldsymbol{c}}_t^{(1_b)} + \boldsymbol{f}_t^{(1_b)} \odot \boldsymbol{c}_{t-1}^{(1_b)} \tag{30}$$

$$\boldsymbol{h}_t^{(1_b)} = \boldsymbol{o}_t^{(1_b)} \odot d(\boldsymbol{c}_t^{(1_b)}) \tag{31}$$

where $\boldsymbol{U}_i^{(1_b)}, \boldsymbol{U}_f^{(1_b)}, \boldsymbol{U}_o^{(1_b)}$, and $\boldsymbol{U}_c^{(1_b)} \in \mathbb{R}^{j \times k}$ and $\boldsymbol{G}_i^{(1_b)}, \boldsymbol{G}_f^{(1_b)}, \boldsymbol{G}_o^{(1_b)}$, and $\boldsymbol{G}_c^{(1_b)} \in \mathbb{R}^{k \times k}$ are the coefficient weight matrices to learn. $\boldsymbol{b}_i^{(1_b)}$, $\boldsymbol{b}_f^{(1_b)}$, $\boldsymbol{b}_o^{(1_b)}$, and $\boldsymbol{b}_c^{(1_b)} \in \mathbb{R}^k$ are the bias vectors. Briefly, (25) and (31) are reformulated as

$$\boldsymbol{h}_t^{(1_f)} = d(\boldsymbol{U}^{(1_f)}\boldsymbol{x}_t + \boldsymbol{G}^{(1_f)}\boldsymbol{h}_{t-1}^{(1_f)} + \boldsymbol{b}^{(1_f)}) \tag{32}$$

$$\boldsymbol{h}_t^{(1_b)} = d(\boldsymbol{U}^{(1_b)}\boldsymbol{x}_t + \boldsymbol{G}^{(1_b)}\boldsymbol{h}_{t+1}^{(1_b)} + \boldsymbol{b}^{(1_b)}) \tag{33}$$

$$\boldsymbol{y}_t^{(1)} = \boldsymbol{V}_y^{(1)}\boldsymbol{h}_t^{(1_f)} + \boldsymbol{V}_y^{(1)}\boldsymbol{h}_t^{(1_b)} + \boldsymbol{b}_y^{(1)} \tag{34}$$

where $\boldsymbol{V}_y^{(1)} \in \mathbb{R}^{2k \times q}$ is the coefficient weight matrix, and $\boldsymbol{b}_y^{(1)} \in \mathbb{R}^q$ is the bias vector. The output of the lower layer is fed to the next subsequent layer (Fig. 4). Thus, the output ($\boldsymbol{y}_t^{(1)}$) of the 1-layered Bi-LSTM is fed to the second layer (i.e., 2-layered Bi-LSTM). Equations (32)–(34) are written below for the 2-layered Bi-LSTM. Note that the input of the 2-layered Bi-LSTM is $\boldsymbol{x}_t = \boldsymbol{y}_t^{(1)}$.

$$\boldsymbol{h}_t^{(2_f)} = d(\boldsymbol{U}^{(2_f)}\boldsymbol{x}_t + \boldsymbol{G}^{(2_f)}\boldsymbol{h}_{t-1}^{(2_f)} + \boldsymbol{b}^{(2_f)}) \tag{35}$$

$$\boldsymbol{h}_t^{(2_b)} = d(\boldsymbol{U}^{(2_b)}\boldsymbol{x}_t + \boldsymbol{G}^{(2_b)}\boldsymbol{h}_{t+1}^{(2_b)} + \boldsymbol{b}^{(2_b)}) \tag{36}$$

$$\boldsymbol{y}_t^{(2)} = \boldsymbol{V}_y^{(2)}\boldsymbol{h}_t^{(2_f)} + \boldsymbol{V}_y^{(2)}\boldsymbol{h}_t^{(2_b)} + \boldsymbol{b}_y^{(2)} \tag{37}$$

where $\boldsymbol{V}_y^{(2)} \in \mathbb{R}^{2k \times q}$ is the coefficient weight matrix, and $\boldsymbol{b}_y^{(2)} \in \mathbb{R}^q$ is the bias vector. Similarly, the n-layered Bi-LSTM is described as follows, where the input is $\boldsymbol{x}_t = \boldsymbol{y}_t^{(n-1)}$:

$$\boldsymbol{h}_t^{(n_f)} = d(\boldsymbol{U}^{(n_f)}\boldsymbol{x}_t + \boldsymbol{G}^{(n_f)}\boldsymbol{h}_{t-1}^{(n_f)} + \boldsymbol{b}^{(n_f)}) \tag{38}$$

$$\boldsymbol{h}_t^{(n_b)} = d(\boldsymbol{U}^{(n_b)}\boldsymbol{x}_t + \boldsymbol{G}^{(n_b)}\boldsymbol{h}_{t+1}^{(n_b)} + \boldsymbol{b}^{(n_b)}) \tag{39}$$

$$\boldsymbol{y}_t^{(n)} = \boldsymbol{V}_y^{(n)}\boldsymbol{h}_t^{(n_f)} + \boldsymbol{V}_y^{(n)}\boldsymbol{h}_t^{(n_b)} + \boldsymbol{b}_y^{(n)} \tag{40}$$

where $\boldsymbol{V}_y^{(n)} \in \mathbb{R}^{2k \times q}$ is the coefficient weight matrix, and $\boldsymbol{b}_y^{(n)} \in \mathbb{R}^q$ is the bias vector. Finally, a softmax layer is added on top of the output layer to model the multi-class probabilities, and described as follows:

$$softmax(\boldsymbol{Y})_i = \frac{e^{\boldsymbol{Y}_i}}{\sum_{j=1}^{C} e^{\boldsymbol{Y}_j}} \tag{41}$$

where C is the number of classes, and numerator $e^{\boldsymbol{Y}_i}$ is the exponential function applied to each element of $\boldsymbol{Y}$. Denominator $\sum_{j=1}^{C} e^{\boldsymbol{Y}_j}$ is the sum of the exponential functions of all elements.

V. Experimental Results

We conducted a series of experiments on seven different datasets and their combination to evaluate the code classification performance of our proposed model. The datasets, evaluation matrices, implementation details, and results are elaborated in detail in the subsequent sections.

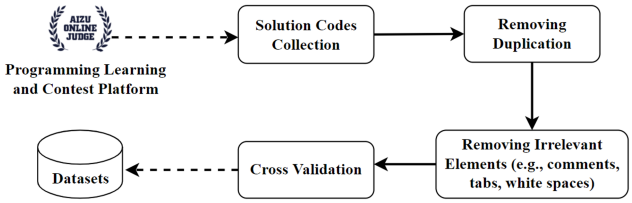


Fig. 5. Overview of the dataset creation phases.

A. Datasets

The datasets used in our experiments are extracted from the Aizu Online Judge (AOJ) system [55], [56]. The growing resources of the AOJ system have been used by various ML/AI-based projects in recent years. For example, IBM and Google DeepMind have used the solution codes of AOJ in their CodeNet [57] and AlphaCode [58] projects, respectively. Fig. 5 provides an overview of the dataset creation using the AOJ system. First, we extract the solution codes from the AOJ based on different algorithms and problem names. Second, duplicate solution codes and irrelevant elements are removed from the codes. Third, a cross-validation of the solution codes is performed to ensure that the codes are in order after removing the irrelevant elements. Finally, the datasets are created.

We use seven different datasets in the experiments, namely Sorting, Searching, Graph & Tree (G&T), Numerical Computation (NC), Basic Data Structures (BDS), and their combinations. The dataset combination is used in the experiments to increase the dataset diversity and complexity. We assign a ground truth value or label to the solution codes based on the algorithm used or the problem name. For example, the solution codes of *Counting Sort* are labeled as *Counting Sort*. The sorting dataset contains solution codes for various sorting algorithms (e.g., *Bubble Sort*, *Insertion Sort*, *Selection Sort*, *Merge Sort*, *Counting Sort*, *Shell Sort*, and *Quick Sort*). Similarly, the Searching, G&T, BDS, and NC datasets contain the solution codes of various algorithms and problems in each category. Table I presents the statistics for these datasets.

B. Evaluation Metrics

In classification spaces, the classifier performance is usually defined by the confusion matrix with respect to the classifier. The accuracy, recall (sensitivity), precision, and F1-score (recall and precision) are calculated from the confusion matrix entries. We follow the standard evaluation approach [19], [49], [59] for our code classification task. First, the multi-class classification accuracy described below is defined as the average number of correct predictions relative to the total number of predictions.

$$\boldsymbol{A} = \frac{1}{N}\sum_{k=1}^{|C|}\sum_{x:f(x)=k} Q(f(x) = \hat{f}(x)) \tag{42}$$

where Q is the function that returns 1 if the class is true and 0, otherwise. C is the number of classes. $f(x) \in C = \{1, 2, 3, \ldots, n\}$.

The program code datasets may contain various data imbalances. Advanced evaluation metrics are adopted to obtain an unbiased performance evaluation for the imbalanced datasets.

TABLE I
STATISTICAL OVERVIEW OF THE DATASETS AND THEIR DISTRIBUTIONS FOR MODEL TRAINING AND VALIDATION

Sl.	Datasets	# Codes	# Programming Languages	Average Length of Code Characters	# Classes	# Train Codes	# Validation Codes	# Test Codes
1	Sorting	53308	12	840.44	8	42646	6930	3732
2	Searching	25994	12	650.22	5	20795	3379	1820
3	NC	30871	12	516.00	6	24696	4013	2162
4	G&T	38761	10	1658.46	14	31008	5039	2714
5	Sorting + Searching	80745	12	801.77	15	64596	10496	5653
6	NC + BDS	54210	12	768.65	11	43368	7047	3795
7	Sorting + Searching + G&T	119476	12	1079.90	29	95580	15532	8364

In addition to the accuracy, precision, recall, and F1-score, we also perform calculations for the macro, micro, and weighted settings. The micro-average is calculated across all samples, summing all true positives (TP), false negatives (FN) and false positives (FP). Thus, the micro-precision ($\boldsymbol{P}_\mu$), recall ($\boldsymbol{R}_\mu$), and F1-score ($\boldsymbol{F1}_\mu$) are calculated by (43)–(45):

$$\boldsymbol{P}_\mu = \frac{\sum_{k=1}^{|C|} TP_k}{\sum_{i=k}^{|C|}(TP_k + FP_k)} \tag{43}$$

$$\boldsymbol{R}_\mu = \frac{\sum_{k=1}^{|C|} TP_k}{\sum_{k=1}^{|C|}(TP_k + FN_k)} \tag{44}$$

$$\boldsymbol{F1}_\mu = \frac{2 \times \boldsymbol{P}_\mu \times \boldsymbol{R}_\mu}{\boldsymbol{P}_\mu + \boldsymbol{R}_\mu} \tag{45}$$

A higher $\boldsymbol{F1}_\mu$ score indicates a better overall performance of the classification model. Therefore, it is not sensitive to individual classes because an imbalanced class data distribution can misrepresent the overall performance of the classification model. However, the macro-average takes into account the performance of individual classes. A higher macro F1-score means a better model performance for the individual classes. If the class data distribution is imbalanced, the macro-average is more appropriate than the micro-average. The macro-precision ($\boldsymbol{P}_\rho$), recall ($\boldsymbol{R}_\rho$), and F1-score ($\boldsymbol{F1}_\rho$) are calculated using the following set of (46)–(48):

$$\boldsymbol{P}_\rho = \frac{1}{|C|}\sum_{k=1}^{|C|}\frac{TP_k}{TP_k + FP_k} = \frac{\sum_{k=1}^{|C|}\boldsymbol{P}_k}{|C|} \tag{46}$$

$$\boldsymbol{R}_\rho = \frac{1}{|C|}\sum_{k=1}^{|C|}\frac{TP_k}{TP_k + FN_k} = \frac{\sum_{k=1}^{|C|}\boldsymbol{R}_k}{|C|} \tag{47}$$

$$\boldsymbol{F1}_\rho = \frac{2 \times \boldsymbol{P}_\rho \times \boldsymbol{R}_\rho}{\boldsymbol{P}_\rho + \boldsymbol{R}_\rho} \tag{48}$$

The weighted average F1-score, on the other hand, is computed from the mean of all F1-scores of the individual classes considering the support of the individual classes. "Support" refers to the number of class instances. The term "weight" refers to the ratio of instances of each class to the sum of all instances. The weighted-precision ($\boldsymbol{P}_\omega$), recall ($\boldsymbol{R}_\omega$), and F1-score ($\boldsymbol{F1}_\omega$) are calculated as follows in (49)–(51):

$$\boldsymbol{P}_\omega = \frac{1}{|S|}\sum_{k=1}^{|C|}\frac{TP_k}{TP_k + FP_k} \times |s_k| = \frac{\sum_{k=1}^{|C|}\boldsymbol{P}_k \times |s_k|}{|S|} \tag{49}$$

$$\boldsymbol{R}_\omega = \frac{1}{|S|}\sum_{k=1}^{|C|}\frac{TP_k}{TP_k + FN_k} \times |s_k| = \frac{\sum_{k=1}^{|C|}\boldsymbol{R}_k \times |s_k|}{|S|} \tag{50}$$

$$\boldsymbol{F1}_\omega = \frac{1}{|S|}\sum_{k=1}^{|C|}\boldsymbol{F1}_k \times |s_k| \tag{51}$$

where $|s_k|$ is the support of the k class, and $|S|$ is the sum of all supports. The Cohen Kappa (κ) [59] score is used to evaluate the classification model's performance and calculated as follows:

$$\kappa = \frac{\phi_o - \phi_e}{1 - \phi_e} \tag{52}$$

where ϕ_o is the observed probability indicating the classification model accuracy (or perfect match), and ϕ_e represents the model prediction and actual class value by random matching. The Area Under Receiver Operating Characteristic Curve (ROC AUC) [60] is employed to evaluate our multi-class classification model. In this context, two strategies are considered for the classification model evaluation: "one-vs-one (OvO)" and "one-vs-rest (OvR)". Here, OvO calculates the average of the pairwise ROC AUC score, and OvR computes the average score for each class against other classes. These results seem particularly optimistic when there is a strong class imbalance, that is, the number of minority class instances is small.

C. Implementation Details

We use different hyperparameters in the proposed n-layered Bi-LSTM model to achieve better results. A program code is usually a collection of complex instructions, including mathematical operations, methods, functions, variables, keywords, and tokens. They are interrelated in a program code. The selection of optimal hyperparameters is greatly important in understanding complex code interrelationships. We employ three ($n = 3$) Bi-LSTM layers as part of the DNN architecture. The Adam [61] is applied as the network optimization method. Different numbers are used for the batch size (β), (i.e., $\beta = \{16, 32, 64\}$) and the learning rate (η) (i.e., $\eta = \{0.001, 0.005, 0.01\}$). The activation functions in the dense layer are sigmoid ($\Lambda(z) = \frac{1}{1+e^{-z}}$), tanh ($\Lambda(z) = \frac{1-e^{-2z}}{1+e^{-2z}}$), and relu ($\Lambda(z) = max(0, z)$). We also conduct experiments with and without a dropout layer, where the dropout layer is placed before the dense layer. The dropout (ξ) values are 0.1 and 0.3. The sparse categorical cross entropy is used as the loss function ($\mathcal{L}$) of the classification model described

TABLE II
HYPERPARAMETER SETTINGS

Hyperparameters	Value
Vocabulary size (v)	10,000
Embedding dimension (d)	200
Maximum code sequence length (m)	500
Padding ($padding\_type$) & Truncating ($trunc\_type$)	post
Out of vocabulary token (oov_tok)	$< OOV >$
Optimization method	adam
Loss function ($cross\_entropy$)	sparse_categorical_crossentropy
Activation functions (Υ) of dense layer	sigmoid, tanh, and ReLU
Epochs ($epoch$)	25 and 50
Number of nodes in Bi-LSTM	400 ($\overrightarrow{200}$ + $\overleftarrow{200}$)
Number of nodes in Dense layer	200
Number of layers of Bi-LSTM	3
Training data	80%
Validation data	13%
Testing data	7%

in (53).

$$\mathcal{L}(w) = -\sum_{c=1}^{C} y_c \cdot log(\hat{y_c}) \tag{53}$$

where w is the model parameter; C is the number of classes; and y_c and $\hat{y_c}$ are the true and predicted labels, respectively. Table II presents the other hyperparameters. All experiments on the proposed classification model are implemented on a Keras+TensorFlow framework with Python 3 and Google Compute Engine backend in Google Colab.

D. Results

Tables III–VI present the quantitative classification results (%) of the macro-precision ($\boldsymbol{P}_\rho$) and recall ($\boldsymbol{R}_\rho$) and the weighted-precision ($\boldsymbol{P}_\omega$) and recall ($\boldsymbol{R}_\omega$) of the proposed n-layered Bi-LSTM and other state-of-the-art models over the seven datasets. The hyperparameter settings (i.e., learning rate $\eta = 0.001$, $epoch = 50$, batch sizes $\beta = \{16, 32, 64\}$, activation functions $\Upsilon = \{\tanh, \textit{sigmoid, relu}\}$, and dropout $\xi = \{none\}$) are applied in these experiments. It is observed that the LSTM model fails to achieve better results compared to the Bi-LSTM and 2- and 3-layered Bi-LSTM models. The LSTM model particularly achieved low $\boldsymbol{P}_\rho$, $\boldsymbol{R}_\rho$, $\boldsymbol{P}_\omega$, and $\boldsymbol{R}_\omega$ scores in the *NC*, *G&T*, and *Sorting + Searching + G&T* datasets. The LSTM model also fails to provide good results with different hyperparameters for the *Searching* dataset. In contrast, the 2-layered Bi-LSTM model achieves better $\boldsymbol{P}_\rho$, $\boldsymbol{R}_\rho$, $\boldsymbol{P}_\omega$, and $\boldsymbol{R}_\omega$ scores compared to the LSTM and Bi-LSTM models.

Figs. 6–9 show the average $\boldsymbol{F1}_\rho$ and $\boldsymbol{F1}_\omega$ scores for each dataset. It can be seen that the 2-layered Bi-LSTM model outperforms the other models, achieving an average $\boldsymbol{F1}_\rho$ score of 96.10 ± 0.60 and an $\boldsymbol{F1}_\omega$ score of 97.10 ± 0.25. The Bi-LSTM model achieves an average $\boldsymbol{F1}_\rho$ score of 95.00 ± 0.45 and an $\boldsymbol{F1}_\omega$ score of 96.20 ± 0.45. The LSTM model yields an average $\boldsymbol{F1}_\rho$ score of 87.00 ± 0.70 and an $\boldsymbol{F1}_\omega$ score of 89.00 ± 0.10 due to the long-term dependencies, diversity, and complex context of the solution codes greatly affecting

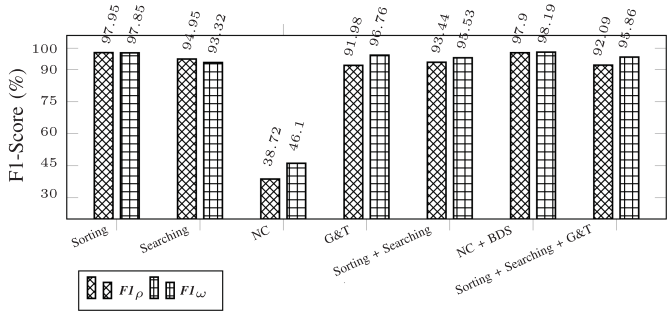


Fig. 6. Average $\boldsymbol{F1}_\rho$ and $\boldsymbol{F1}_\omega$ scores using the LSTM model.

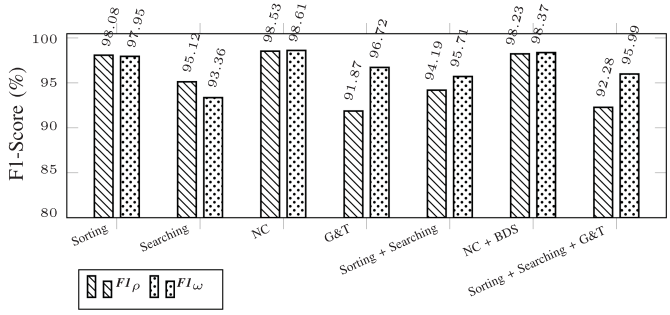


Fig. 7. Average $\boldsymbol{F1}_\rho$ and $\boldsymbol{F1}_\omega$ scores using the Bi-LSTM model.

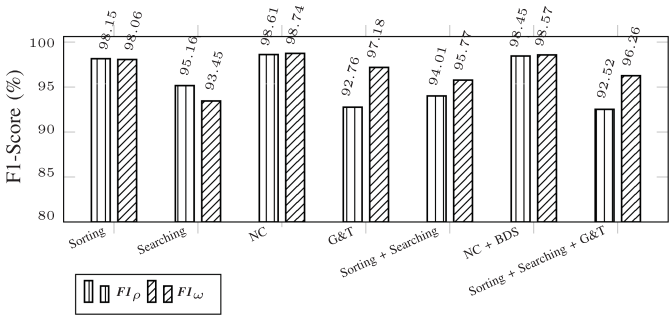


Fig. 8. Average $\boldsymbol{F1}_\rho$ and $\boldsymbol{F1}_\omega$ scores using the 2-layered Bi-LSTM model.

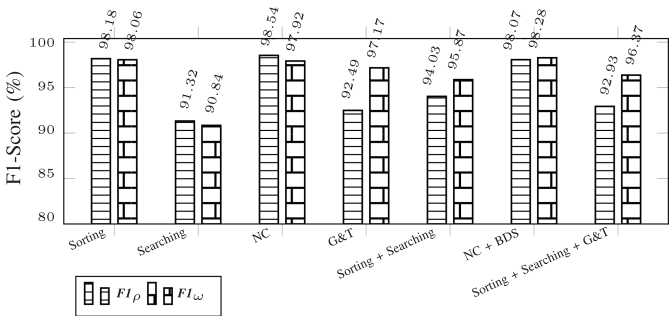


Fig. 9. Average $\boldsymbol{F1}_\rho$ and $\boldsymbol{F1}_\omega$ scores using the 3-layered Bi-LSTM model.

the LSTM performance. The Bi-LSTM solves the problems to some extent and significantly improves the classification results. However, this performance is worse than that of the layered Bi-LSTM. The LSTM and Bi-LSTM models struggle to produce better results compared to the layered Bi-LSTM models on the *G&T*, *Sorting + Searching*, and *Sorting + Searching +*

TABLE III
QUANTITATIVE CLASSIFICATION RESULTS (%) OF THE MACRO- $\boldsymbol{P}_\rho$ AND $\boldsymbol{R}_\rho$ AND THE WEIGHTED- $\boldsymbol{P}_\omega$ AND $\boldsymbol{R}_\omega$ USING THE LSTM

Datasets	β	Macro-precision and recall						Weighted-precision and recall					
		$\Upsilon \longrightarrow Sigmoid$		$\Upsilon \longrightarrow Tanh$		$\Upsilon \longrightarrow ReLU$		$\Upsilon \longrightarrow Sigmoid$		$\Upsilon \longrightarrow Tanh$		$\Upsilon \longrightarrow ReLU$	
		$\boldsymbol{P}_\rho$	$\boldsymbol{R}_\rho$	$\boldsymbol{P}_\rho$	$\boldsymbol{R}_\rho$	$\boldsymbol{P}_\rho$	$\boldsymbol{R}_\rho$	$\boldsymbol{P}_\omega$	$\boldsymbol{R}_\omega$	$\boldsymbol{P}_\omega$	$\boldsymbol{R}_\omega$	$\boldsymbol{P}_\omega$	$\boldsymbol{R}_\omega$
Sorting	16	98.21	97.80	98.20	97.88	98.16	97.94	97.94	97.93	98.00	97.99	97.94	97.93
	32	98.18	97.94	97.93	97.62	98.19	97.92	97.94	97.93	97.67	97.66	98.00	97.99
	64	98.19	98.08	98.09	97.78	97.98	97.31	98.10	98.09	97.82	97.82	97.39	97.34
Searching	16	-	-	-	-	94.88	94.57	-	-	-	-	93.01	92.96
	32	95.31	95.65	95.12	95.42	-	-	94.07	94.06	93.62	93.51	-	-
	64	94.51	93.77	-	-	-	-	92.79	92.14	-	-	-	-
NC	16	98.55	98.02	22.84	16.76	22.84	16.76	98.34	98.33	21.59	37.09	21.59	37.09
	32	97.91	98.58	22.84	16.76	96.16	97.47	98.39	98.38	21.59	37.09	97.49	97.45
	64	22.84	16.76	22.84	16.76	22.84	16.76	21.59	37.09	21.59	37.09	21.59	37.09
G&T	16	93.04	92.56	92.60	91.85	92.88	92.80	96.86	96.83	96.98	96.90	97.11	97.05
	32	91.92	91.07	91.42	91.24	92.65	92.02	96.41	96.38	96.53	96.42	97.00	96.97
	64	92.05	92.17	90.93	90.51	92.65	92.94	96.91	96.86	96.22	96.05	97.45	97.34
Sorting + Searching	16	93.64	93.19	93.93	93.22	94.00	92.63	95.68	95.64	95.52	95.50	95.49	95.41
	32	93.89	93.49	94.39	93.21	93.82	92.93	95.44	95.41	95.69	95.70	95.42	95.43
	64	94.11	92.90	93.88	93.88	93.83	92.73	95.73	95.71	95.54	95.52	95.65	95.61
NC + BDS	16	98.38	97.79	98.17	97.60	98.09	97.49	98.26	98.26	98.21	98.20	98.10	98.07
	32	98.14	97.52	98.06	97.41	98.34	97.70	98.20	98.18	98.13	98.12	98.28	98.25
	64	98.03	97.45	98.46	97.68	98.45	97.66	98.05	98.05	98.38	98.36	98.27	98.26
Sorting + Searching + G&T	16	91.97	91.44	92.91	91.77	92.26	91.77	95.70	95.63	95.91	95.91	96.13	96.10
	32	93.15	92.41	92.41	91.45	92.70	91.77	96.09	96.06	96.03	96.04	96.04	96.05
	64	92.72	92.16	92.65	91.95	91.98	91.42	95.70	95.63	95.84	95.82	95.65	95.66

TABLE IV
QUANTITATIVE CLASSIFICATION RESULTS (%) OF THE MACRO- $\boldsymbol{P}_\rho$ AND $\boldsymbol{R}_\rho$ AND THE WEIGHTED- $\boldsymbol{P}_\omega$ AND $\boldsymbol{R}_\omega$ USING THE BI-LSTM

Datasets	β	Macro-precision and recall						Weighted-precision and recall					
		$\Upsilon \longrightarrow Sigmoid$		$\Upsilon \longrightarrow Tanh$		$\Upsilon \longrightarrow ReLU$		$\Upsilon \longrightarrow Sigmoid$		$\Upsilon \longrightarrow Tanh$		$\Upsilon \longrightarrow ReLU$	
		$\boldsymbol{P}_\rho$	$\boldsymbol{R}_\rho$	$\boldsymbol{P}_\rho$	$\boldsymbol{R}_\rho$	$\boldsymbol{P}_\rho$	$\boldsymbol{R}_\rho$	$\boldsymbol{P}_\omega$	$\boldsymbol{R}_\omega$	$\boldsymbol{P}_\omega$	$\boldsymbol{R}_\omega$	$\boldsymbol{P}_\omega$	$\boldsymbol{R}_\omega$
Sorting	16	98.19	97.97	98.39	98.01	98.19	98.04	97.92	97.91	98.11	98.09	98.00	97.99
	32	98.31	97.95	98.40	97.92	97.96	97.58	98.00	97.99	98.11	98.09	97.58	97.56
	64	98.40	98.12	97.85	97.84	98.40	98.17	98.11	98.09	97.77	97.77	98.19	98.17
Searching	16	95.53	95.40	95.09	94.56	95.22	95.18	93.70	93.68	93.24	93.13	93.34	93.35
	32	94.94	95.12	95.20	95.32	95.15	95.07	93.31	93.24	93.46	93.46	93.25	93.18
	64	94.85	94.91	95.50	95.24	95.31	94.78	93.27	93.24	93.68	93.62	93.39	93.35
NC	16	98.94	98.63	98.79	98.31	98.98	98.63	98.85	98.84	98.43	98.42	98.70	98.70
	32	98.72	98.21	98.58	98.78	98.73	98.67	98.56	98.56	98.80	98.79	98.80	98.79
	64	98.94	98.50	97.66	97.69	98.53	98.44	98.75	98.75	98.15	98.14	98.52	98.51
G&T	16	91.33	91.44	91.80	91.44	92.64	91.73	96.89	96.79	96.66	96.61	96.65	96.64
	32	91.92	91.82	92.03	92.13	92.47	91.84	96.85	96.75	96.96	96.86	96.83	96.75
	64	92.76	92.33	93.35	92.75	90.94	90.58	96.88	96.79	96.78	96.72	96.61	96.53
Sorting + Searching	16	93.87	93.98	94.53	94.63	94.80	94.45	95.86	95.80	95.76	95.73	95.69	95.64
	32	94.00	93.25	94.27	94.18	94.63	94.39	95.70	95.70	95.64	95.59	95.91	95.89
	64	95.15	94.71	93.62	94.06	94.24	93.37	95.88	95.82	95.73	95.68	95.61	95.55
NC + BDS	16	98.38	98.20	98.49	97.87	98.28	98.02	98.40	98.39	98.40	98.39	98.29	98.28
	32	98.76	98.17	98.49	97.79	98.49	97.89	98.55	98.55	98.32	98.31	98.35	98.33
	64	98.61	97.83	98.56	97.94	98.46	98.11	98.33	98.31	98.35	98.33	98.48	98.47
Sorting + Searching + G&T	16	92.85	92.42	92.92	92.78	91.67	91.68	96.21	96.19	96.23	96.15	95.78	95.74
	32	92.83	91.90	92.17	91.87	93.02	92.09	96.03	96.03	95.95	95.89	96.08	96.09
	64	92.58	92.09	92.60	91.40	93.54	92.33	96.02	95.97	95.89	95.92	96.06	96.04

G&T datasets. The deep layered architecture enables the layered Bi-LSTM model to better learn the complex context of large diverse codes. Considering the same hyperparameter settings, the $\boldsymbol{A}$, κ, and OvR AUC ROC scores for all models over the seven datasets are also calculated (Table VII). The results show that the 2-layered Bi-LSTM model outperforms the other models with an average $\boldsymbol{A}$ of 97.00 ± 0.10, κ of 96.29, and OvR of 99.68. Although the 3-layered Bi-LSTM model achieves relatively better results ($\boldsymbol{A}$ of 96.30 ± 0.07) than the LSTM ($\boldsymbol{A}$ of 90.19 ± 0.50), it is not as good as the 2-layered Bi-LSTM. These results show that the 2-layered Bi-LSTM model is more suitable for the current datasets. Thus, we avoid 3- and more-layered Bi-LSTM models in further experiments.

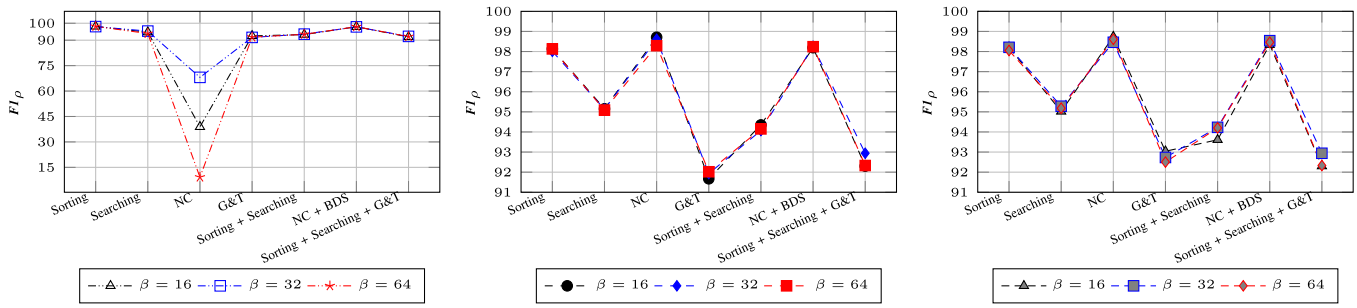
We evaluate the model performance using different hyperparameters (e.g., β, Υ, η, and ξ), as shown in Figs. 10, 11, 12, and 13. The experiments provide various additional insights. Figs. 10, 11, and 13 depict that hyperparameters (β, Υ, and ξ) only have light effects on the $\boldsymbol{F1}_\rho$ score when considered. Similar performance trends observed in the calculated $\boldsymbol{A}$ and $\boldsymbol{F1}_\omega$ scores. By contrast, a significant impact on the $\boldsymbol{F1}_\rho$ score is found when different learning rates (e.g., $\eta = \{0.001, 0.005, 0.01\}$) are considered for all the models across the seven datasets, as shown in Fig. 12. The $\boldsymbol{F1}_\rho$ scores are significantly improved when η is set to 0.001 compared to $\eta = 0.005$ and $\eta = 0.01$, implying that slowing down the η improves the model performance. The average $\boldsymbol{F1}_\rho$ scores for

TABLE V
QUANTITATIVE CLASSIFICATION RESULTS (%) OF THE MACRO- $\boldsymbol{P}_\rho$ AND $\boldsymbol{R}_\rho$ AND THE WEIGHTED- $\boldsymbol{P}_\omega$ AND $\boldsymbol{R}_\omega$ USING THE 2-LAYERED BI-LSTM

Datasets	β	Macro-precision and recall						Weighted-precision and recall					
		$\Upsilon \longrightarrow Sigmoid$		$\Upsilon \longrightarrow Tanh$		$\Upsilon \longrightarrow ReLU$		$\Upsilon \longrightarrow Sigmoid$		$\Upsilon \longrightarrow Tanh$		$\Upsilon \longrightarrow ReLU$	
		$\boldsymbol{P}_\rho$	$\boldsymbol{R}_\rho$	$\boldsymbol{P}_\rho$	$\boldsymbol{R}_\rho$	$\boldsymbol{P}_\rho$	$\boldsymbol{R}_\rho$	$\boldsymbol{P}_\omega$	$\boldsymbol{R}_\omega$	$\boldsymbol{P}_\omega$	$\boldsymbol{R}_\omega$	$\boldsymbol{P}_\omega$	$\boldsymbol{R}_\omega$
Sorting	16	98.45	98.17	98.14	97.88	98.48	98.16	98.22	98.20	97.95	97.93	98.16	98.15
	32	98.29	98.15	98.21	97.96	98.45	98.21	98.13	98.12	98.05	98.04	98.21	98.20
	64	98.08	97.86	98.15	98.02	98.23	97.97	97.90	97.88	98.05	98.04	98.02	98.01
Searching	16	94.95	95.11	95.04	94.78	95.21	95.12	93.23	93.18	93.61	93.51	93.53	93.51
	32	95.33	94.84	95.80	95.50	95.21	95.08	93.43	93.35	93.95	93.90	93.43	93.40
	64	94.45	95.25	95.01	94.71	95.58	95.13	93.63	93.57	92.99	92.96	93.71	93.62
NC	16	98.85	98.61	98.83	98.66	98.79	98.75	98.89	98.88	98.80	98.79	98.89	98.88
	32	98.33	98.52	98.52	98.56	98.76	98.13	98.71	98.70	98.75	98.75	98.66	98.65
	64	98.98	98.23	98.46	98.56	98.90	98.57	98.57	98.56	98.71	98.70	98.75	98.75
G&T	16	93.17	92.92	93.49	92.94	92.86	92.32	97.34	97.34	97.51	97.49	97.38	97.27
	32	93.08	93.11	92.70	92.20	93.29	92.54	97.27	97.23	97.27	97.12	97.24	97.23
	64	91.74	91.67	93.64	93.24	92.80	92.28	96.48	96.42	97.43	97.38	97.17	97.12
Sorting + Searching	16	94.17	93.36	93.71	93.36	94.04	93.41	95.85	95.82	95.65	95.63	95.53	95.50
	32	94.16	94.84	94.12	94.12	94.37	94.26	96.05	95.94	95.52	95.48	96.13	96.09
	64	93.79	94.21	94.87	94.44	94.48	93.83	95.77	95.70	95.92	95.89	95.92	95.89
NC + BDS	16	98.40	98.03	98.57	98.01	98.97	98.13	98.40	98.39	98.39	98.39	98.62	98.60
	32	98.81	98.35	98.70	98.20	98.87	98.36	98.71	98.70	98.50	98.49	98.71	98.70
	64	98.72	98.07	98.76	98.28	98.72	98.26	98.53	98.52	98.69	98.68	98.63	98.62
Sorting + Searching + G&T	16	92.82	92.17	92.66	92.58	92.41	91.77	96.22	96.22	96.19	96.10	96.11	96.05
	32	93.26	92.86	93.18	92.81	93.53	92.33	96.41	96.37	96.33	96.35	96.36	96.35
	64	92.51	92.03	92.64	92.10	92.90	92.32	96.46	96.48	96.26	96.22	96.41	96.40

TABLE VI
QUANTITATIVE CLASSIFICATION RESULTS (%) OF THE MACRO- $\boldsymbol{P}_\rho$ AND $\boldsymbol{R}_\rho$ AND THE WEIGHTED- $\boldsymbol{P}_\omega$ AND $\boldsymbol{R}_\omega$ USING THE 3-LAYERED BI-LSTM

Datasets	β	Macro-precision and recall						Weighted-precision and recall					
		$\Upsilon \longrightarrow Sigmoid$		$\Upsilon \longrightarrow Tanh$		$\Upsilon \longrightarrow ReLU$		$\Upsilon \longrightarrow Sigmoid$		$\Upsilon \longrightarrow Tanh$		$\Upsilon \longrightarrow ReLU$	
		$\boldsymbol{P}_\rho$	$\boldsymbol{R}_\rho$	$\boldsymbol{P}_\rho$	$\boldsymbol{R}_\rho$	$\boldsymbol{P}_\rho$	$\boldsymbol{R}_\rho$	$\boldsymbol{P}_\omega$	$\boldsymbol{R}_\omega$	$\boldsymbol{P}_\omega$	$\boldsymbol{R}_\omega$	$\boldsymbol{P}_\omega$	$\boldsymbol{R}_\omega$
Sorting	16	98.28	98.03	98.31	97.86	98.35	97.92	98.10	98.09	98.04	98.04	98.03	98.01
	32	98.60	98.25	98.49	98.25	98.49	98.09	98.27	98.25	98.24	98.23	98.13	98.12
	64	98.06	97.96	98.36	97.99	98.31	97.95	97.89	97.88	98.05	98.04	98.05	98.04
Searching	16	92.80	92.99	92.06	89.73	91.88	89.99	91.34	91.28	90.28	90.13	90.57	90.35
	32	94.72	95.94	93.43	92.86	93.31	95.75	93.51	93.48	93.54	93.55	93.81	93.67
	64	88.71	87.37	92.06	87.07	90.36	86.09	89.16	88.76	89.29	88.65	87.96	87.39
NC	16	98.96	98.66	97.24	98.01	99.38	98.98	98.89	98.88	98.15	98.10	99.12	99.12
	32	98.83	98.53	98.61	98.76	99.03	98.59	98.66	98.65	98.84	98.84	98.84	98.84
	64	97.63	98.32	98.11	98.55	99.06	98.64	98.44	98.42	98.61	98.61	98.80	98.79
G&T	16	92.77	92.71	93.55	93.07	92.58	92.05	97.45	97.42	97.29	97.23	97.30	97.31
	32	93.03	92.61	92.58	92.06	92.93	92.95	97.09	97.08	97.04	96.83	97.37	97.31
	64	92.42	91.80	92.62	92.02	92.25	92.29	97.29	97.31	97.10	97.08	97.02	96.97
Sorting + Searching	16	94.03	93.46	94.60	93.63	93.80	94.23	95.74	95.70	95.89	95.89	96.00	96.00
	32	94.87	93.88	93.84	94.55	93.90	94.16	95.87	95.84	96.00	95.89	95.88	95.82
	64	94.78	93.81	94.36	93.95	94.15	93.88	96.14	96.12	95.99	95.98	95.58	9554
NC + BDS	16	97.61	97.05	98.03	97.98	98.17	97.56	97.67	97.60	98.35	98.33	98.02	98.02
	32	98.72	97.74	98.88	98.22	98.82	97.93	98.45	98.44	98.61	98.60	98.48	98.47
	64	98.07	97.62	98.54	97.75	98.92	98.05	98.26	98.23	98.31	98.28	98.61	98.60
Sorting + Searching + G&T	16	92.82	92.13	92.98	92.33	93.42	92.47	96.05	95.95	96.26	96.19	96.34	96.34
	32	93.90	92.61	93.39	92.21	93.46	92.87	96.46	96.46	96.56	96.56	96.31	96.30
	64	93.86	92.50	93.71	93.12	94.10	92.90	96.40	96.42	96.73	96.68	96.56	95.56

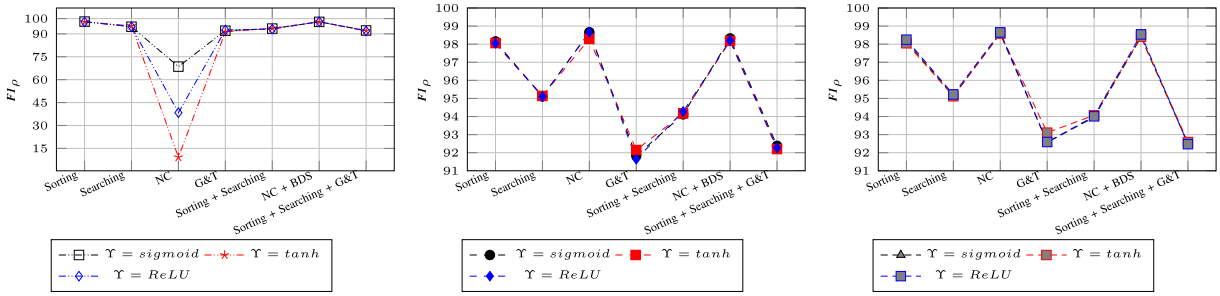


(a) LSTM (b) Bi-LSTM (c) 2-layered Bi-LSTM

Fig. 10. Comparison of the $\boldsymbol{F1}_\rho$ scores with different batch sizes (β) when $\Upsilon = \{sigmoid, tanh, relu\}$, $epoch = 50$, $\xi = none$, and $\eta = 0.001$.

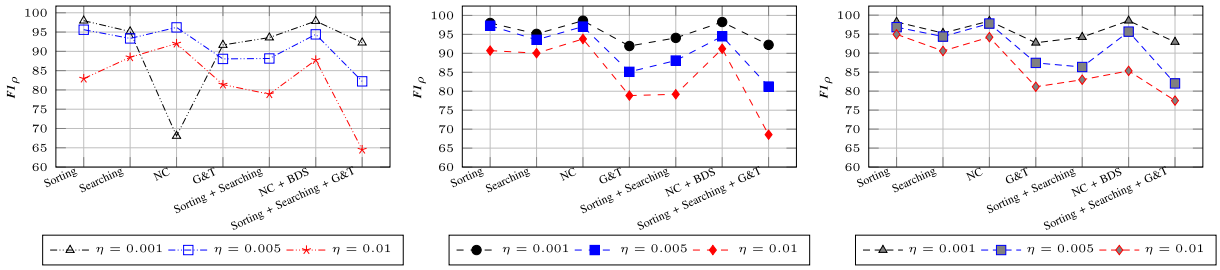
TABLE VII
QUANTITATIVE CLASSIFICATION RESULTS (%) OF THE ACCURACY, KAPPA, AND ROC AUC SCORES USING THE LSTM, BI-LSTM, 2-LAYERED BI-LSTM, AND 3-LAYERED BI-LSTM MODELS

Datasets	Υ	LSTM			Bi-LSTM			2-layered Bi-LSTM			3-layered Bi-LSTM		
		$\boldsymbol{A}$	κ	OvR	$\boldsymbol{A}$	κ	OvR	$\boldsymbol{A}$	κ	OvR	$\boldsymbol{A}$	κ	OvR
Sorting	Sigmoid	97.98	97.60	99.77	98.00	97.60	99.75	98.06	97.69	99.78	98.03	97.65	99.81
	Tanh	97.82	97.40	99.75	97.98	97.59	99.76	98.00	97.62	99.79	98.10	97.73	99.79
	ReLU	97.75	97.32	99.73	97.90	97.50	99.75	98.12	97.76	99.76	98.06	97.68	99.77
Searching	Sigmoid	93.10	90.92	98.95	93.39	91.29	99.30	93.57	91.26	98.91	91.17	87.99	98.02
	Tanh	93.51	91.46	98.96	93.40	91.30	98.96	93.90	91.38	99.03	90.78	87.42	97.85
	ReLU	92.96	90.72	99.00	93.29	91.16	98.78	93.95	91.45	98.99	90.47	87.02	97.97
NC	Sigmoid	77.93	65.26	83.29	98.72	98.31	99.89	98.71	98.31	99.92	98.65	98.23	99.91
	Tanh	37.09	0.00	50.00	98.45	97.97	99.88	98.75	98.35	99.91	98.52	98.05	99.91
	ReLU	57.21	32.28	66.66	98.67	98.25	99.88	98.76	98.37	99.94	98.92	98.57	99.94
G&T	Sigmoid	96.69	96.27	99.70	96.78	96.37	99.72	96.97	96.59	99.78	97.27	96.92	99.79
	Tanh	96.46	96.01	99.67	96.73	96.31	99.70	97.33	96.99	99.75	97.05	96.67	99.76
	ReLU	97.12	96.75	99.36	96.64	96.21	99.39	97.21	96.85	99.67	97.20	96.96	99.78
Sorting + Searching	Sigmoid	95.59	95.14	99.68	95.70	95.34	99.65	95.94	95.39	99.65	95.89	95.46	99.63
	Tanh	95.57	95.12	99.62	95.59	95.22	99.69	95.89	95.22	99.72	95.92	95.50	99.72
	ReLU	95.48	95.02	99.54	95.55	95.26	99.53	96.09	95.40	99.65	95.79	95.35	99.65
NC + BDS	Sigmoid	98.16	97.91	99.75	98.42	98.20	99.80	98.54	98.34	99.83	98.09	97.83	99.78
	Tanh	98.23	97.99	99.7	98.34	98.12	99.81	98.52	98.32	99.83	98.40	98.19	99.76
	ReLU	98.20	97.96	99.77	98.36	98.14	99.82	98.64	98.56	99.84	98.36	98.14	99.80
Sorting + Searching + G&T	Sigmoid	95.77	95.53	99.78	96.06	95.83	99.82	96.48	96.15	99.82	96.28	96.06	99.80
	Tanh	95.92	95.69	99.80	95.89	95.75	99.82	96.25	95.97	99.84	96.48	96.27	99.86
	ReLU	95.94	95.70	99.72	95.74	95.72	99.73	96.40	96.05	99.77	94.40	96.19	99.76
Average		**90.69**	**86.57**	**94.87**	**96.64**	**96.07**	**99.64**	**96.96**	**96.29**	**99.68**	**96.37**	**95.71**	**99.53**



(a) LSTM (b) Bi-LSTM (c) 2-layered Bi-LSTM

Fig. 11. Comparison of the $\boldsymbol{F1}_\rho$ scores with different activation functions (Υ) when $\beta = \{16, 32, 64\}$, $epoch = 50$, $\xi = none$, and $\eta = 0.001$.

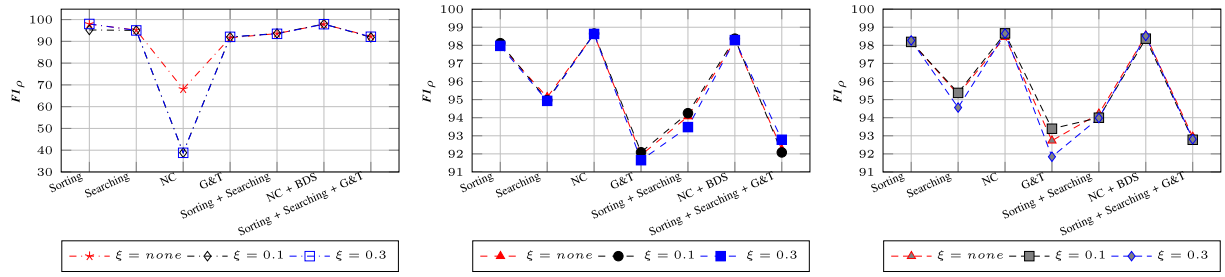


(a) LSTM (b) Bi-LSTM (c) 2-layered Bi-LSTM

Fig. 12. Comparison of the $\boldsymbol{F1}_\rho$ scores with different learning rates (η) when $\beta = 32$, $\Upsilon = \{sigmoid, tanh, relu\}$, $epoch = 50$, and $\xi = none$.

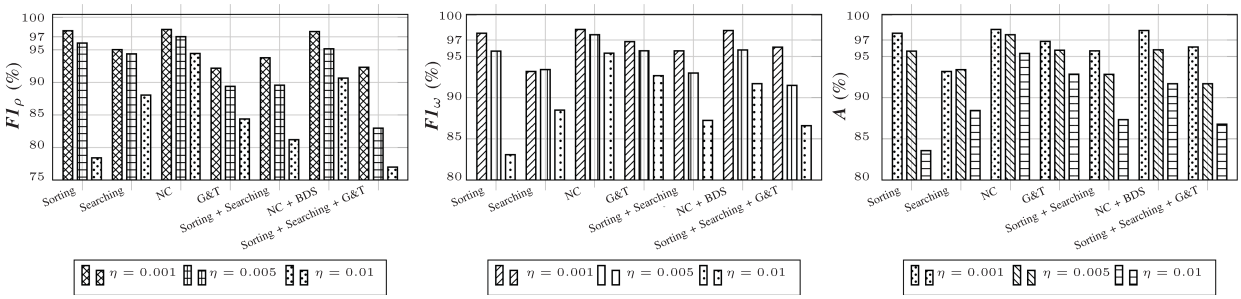
the 2-layered Bi-LSTM model are 95.77, 91.50, and 86.31 with η values of 0.001, 0.005, and 0.01, respectively.

Figs. 14, 15, and 16 compare the $\boldsymbol{F1}_\rho$, $\boldsymbol{F1}_\omega$, and $\boldsymbol{A}$ scores based on a set of hyperparameters for the LSTM, Bi-LSTM, and 2-layered Bi-LSTM, respectively. The results illustrate that the learning rate (η) significantly affects the $\boldsymbol{F1}_\rho$, $\boldsymbol{F1}_\omega$, and $\boldsymbol{A}$ scores with a specific set of hyperparameters ($\beta = 32$, $\Upsilon = relu$, $epoch = 50$, and $\xi = none$). Note that $\boldsymbol{F1}_\mu$ and $\boldsymbol{A}$ produce the same results due to the similar evaluation criteria for the classification model. Therefore, only $\boldsymbol{A}$ is considered during the evaluation and $\boldsymbol{F1}_\mu$ is excluded. Considering the diversity, data volume, and complexity of the *Sorting + Searching + G&T* dataset, the 2-layered Bi-LSTM model achieves better $\boldsymbol{F1}_\rho$, $\boldsymbol{F1}_\omega$, and $\boldsymbol{A}$ scores than the other two models when $\eta = 0.001$ with other hyperparameters. These results indicate that the 2-layered Bi-LSTM model outperforms the LSTM and



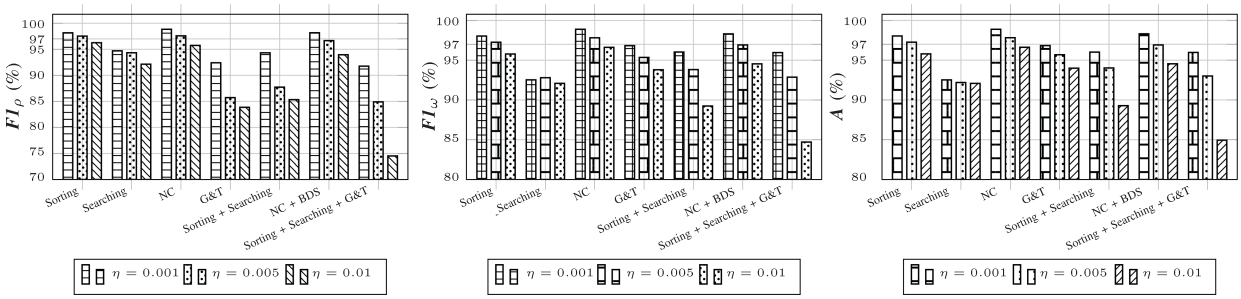
(a) LSTM (b) Bi-LSTM (c) 2-layered Bi-LSTM

Fig. 13. Comparison of the $\boldsymbol{F1}_\rho$ scores with different dropouts (ξ) when $\beta = 32$, $\Upsilon = \{sigmoid, tanh, relu\}$, $epoch = 50$, and $\eta = 0.001$.



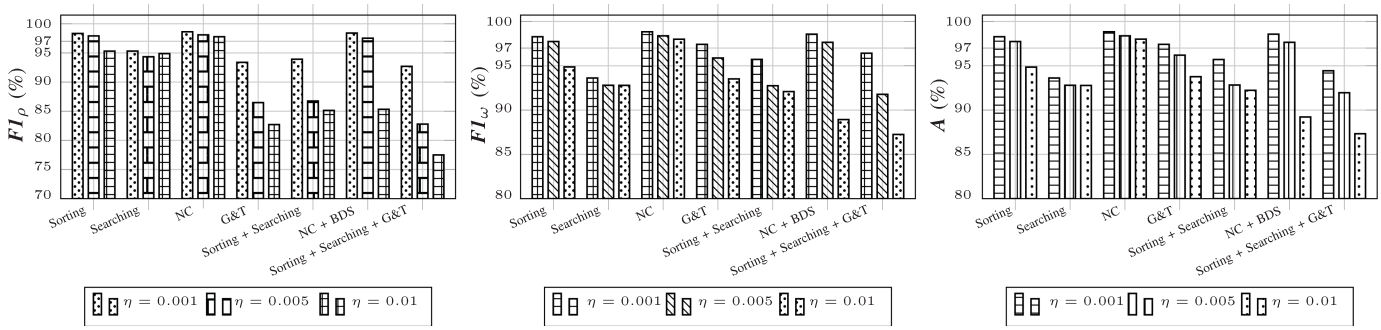
(a) Comparison of $\boldsymbol{F1}_\rho$ scores (b) Comparison of $\boldsymbol{F1}_\omega$ scores (c) Comparison of $\boldsymbol{A}$ scores

Fig. 14. $\boldsymbol{F1}_\rho$, $\boldsymbol{F1}_\omega$, and $\boldsymbol{A}$ scores for the LSTM model with the set of hyperparameters $\eta = \{0.001, 0.005, 0.01\}$, $\beta = 32$, $\Upsilon = relu$, $epoch = 50$, and $\xi = none$.



(a) Comparison of $\boldsymbol{F1}_\rho$ scores (b) Comparison of $\boldsymbol{F1}_\omega$ scores (c) Comparison of $\boldsymbol{A}$ scores

Fig. 15. $\boldsymbol{F1}_\rho$, $\boldsymbol{F1}_\omega$, and $\boldsymbol{A}$ scores for the Bi-LSTM model with the set of hyperparameters $\eta = \{0.001, 0.005, 0.01\}$, $\beta = 32$, $\Upsilon = relu$, $epoch = 50$, and $\xi = none$.



(a) Comparison of $\boldsymbol{F1}_\rho$ scores (b) Comparison of $\boldsymbol{F1}_\omega$ scores (c) Comparison of $\boldsymbol{A}$ scores

Fig. 16. $\boldsymbol{F1}_\rho$, $\boldsymbol{F1}_\omega$, and $\boldsymbol{A}$ scores for the 2-layered Bi-LSTM model with the set of hyperparameters $\eta = \{0.001, 0.005, 0.01\}$, $\beta = 32$, $\Upsilon = relu$, $epoch = 50$, and $\xi = none$.

TABLE VIII
QUANTITATIVE RESULTS (%) FOR CLASSIFICATION WITH THE SOTA MODELS

Datasets	P_ρ	R_ρ	$F1_\rho$	P_ω	R_ω	$F1_\omega$	A
GRU	92.79	92.21	92.45	95.87	95.82	95.82	95.82
Bi-GRU	93.21	92.66	92.91	95.96	95.95	95.95	95.95
LSTM with Attention	93.20	92.47	92.76	96.23	96.20	96.19	95.20
2-layered Bi-LSTM	92.90	92.32	92.57	96.41	96.40	96.39	96.40
3-layered Bi-LSTM	94.10	92.90	93.42	96.56	96.56	96.55	96.56

Bi-LSTM models on all three evaluation metrics $\boldsymbol{F1}_\rho$, $\boldsymbol{F1}_\omega$, and $\boldsymbol{A}$ across the seven datasets.

Furthermore, we conducted additional experiments using state-of-the-art (SOTA) models, specifically the Gated Recurrent Unit (GRU), Bidirectional GRU (Bi-GRU), and LSTM with Attention, employing the *Sorting + Searching + G&T* dataset. For consistency across all SOTA models, we set the hyperparameter values as follows: $epoch = 50$, $\Upsilon =$ ReLu, $\beta = 64$, $\xi =$ none, and $\eta = 0.001$. The results of these experiments are presented in Table VIII. Notably, the LSTM with Attention model achieved $\boldsymbol{F1}_\rho$, $\boldsymbol{F1}_\omega$, and $\boldsymbol{A}$ scores of 92.76%, 96.19%, and 96.20%, respectively, surpassing the performance of the other SOTA models.

However, it is worth mentioning that even though the LSTM with Attention model performed well, it still fell short when compared to the layered Bi-LSTM models. The 3-layered Bi-LSTM model, in particular, outperformed all other SOTA models, achieving $\boldsymbol{F1}_\rho$, $\boldsymbol{F1}_\omega$, and $\boldsymbol{A}$ scores of 93.42%, 96.55%, and 96.56%, respectively. These results further underscore the effectiveness of the proposed layered Bi-LSTM model compared to SOTA models.

VI. DISCUSSION

A. Performance Analysis

We propose herein an $n-$layered Bi-LSTM model for the program code classification task that takes into account the complex context and diversity of program codes. We evaluate the model performance by training, validating, and testing the model using real-world program codes. The quantitative classification results (Tables III–VI) show that the layered Bi-LSTM model achieves better classification results compared to the LSTM and Bi-LSTM models. Table VII illustrates that the 2-layered Bi-LSTM model achieves approximately 0.32% and 6.27% higher $\boldsymbol{A}$ compared to the Bi-LSTM and LSTM models, respectively. In addition to $\boldsymbol{A}$, the 2-layered Bi-LSTM model also obtains higher κ and OvR scores. However, despite the good performance of these models, a notable performance difference is observed between the $\boldsymbol{F1}_\rho$ and $\boldsymbol{F1}_\omega$ scores for most datasets (Figs. 6–9). Underlying this difference is the class imbalance in the test data. That is, some classes contain many instances, while some have relatively few instances. Fig. 17 presents the computation time (in second) for the model training for each dataset. It is seen that the 3-layered Bi-LSTM model takes more time than the other models. In other words, increasing the number of layers requires more time for model training. However, the overall classification results of the proposed 2-layered Bi-LSTM model fully reflect its superiority over the other state-of-the-art models.

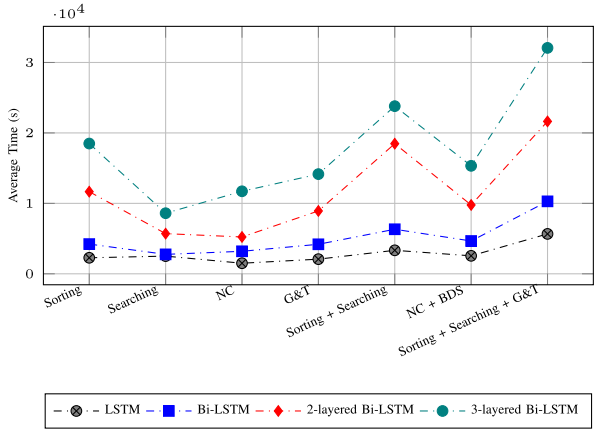


Fig. 17. Computational time comparison for model training when $\beta = \{16, 32, 64\}$, $\Upsilon = relu$, $epoch = 50$, $\xi = none$, and $\eta = 0.001$.

B. Impact of Hyperparameters

The DNN model performance is highly dependent on the optimal hyperparameters. Choosing the optimal parameters is a non-trivial task because it requires heavy parameter fine-tuning. We utilize different hyperparameter sets during the model training and evaluation to achieve better results. Tables III–VII present the model performances based on different hyperparameter settings. Figs. 10–13 depict the classification results of the LSTM, Bi-LSTM, and 2-layered Bi-LSTM models based on various values of parameters β, Υ, η, and ξ. In particular, Fig. 12(c) shows the effects of the η changes on the model performance. The 2-layered Bi-LSTM model exhibits an improved performance of approximately 4.27% with $\eta = 0.001$ when compared to that with $\eta = 0.005$ and approximately 9.46% with $\eta = 0.001$ when compared with $\eta = 0.01$. On the other hand, Figs. 10, 11, and 13 illustrate the limited effect of the parameters on the model performance across all datasets. Figs. 14-16 show the comparative results of the $\boldsymbol{F1}_\rho$, $\boldsymbol{F1}_\omega$, and $\boldsymbol{A}$ scores using η values of 0.001, 0.005, and 0.01, respectively, with a specific hyperparameter set. The results demonstrate the importance of selecting optimal hyperparameters for the model performance.

C. Scalability of the $n-$layered Bi-LSTM Model

We observe the model performance by combining three different datasets to prepare the *Sorting + Searching + G&T* dataset to ensure higher complexity, data size, and diversity. Table VII shows that the 2-layered Bi-LSTM, 3-layered Bi-LSTM, Bi-LSTM, and LSTM models achieve $\boldsymbol{A}$ scores of 96.48%, 96.28%, 96.06%, and 95.77%, respectively, for the *Sorting + Searching + G&T* dataset with a $\Upsilon \rightarrow Sigmoid$. Similar trends are observed for the *Sorting + Searching* dataset, where the 2- and 3-layered Bi-LSTM models obtain higher $\boldsymbol{A}$ scores. This is due to the fact that the layered Bi-LSTM model considers a higher number of

TABLE IX
TRAINABLE PARAMETERS FOR THE MODEL TRAINING

Datasets	Number of trainable parameters in Million			
	LSTM	Bi-LSTM	2-layered Bi-LSTM	3-layered Bi-LSTM
Sorting	2.363	2.723	3.685	4.647
Searching	2.362	2.723	3.684	4.646
NC	2.362	2.723	3.685	4.646
G&T	2.364	2.725	3.686	4.648
Sorting + Searching	2.364	2.725	3.686	4.648
NC + BDS	2.363	2.724	3.685	4.647
Sorting + Searching + G&T	2.367	2.728	3.689	4.651
Average	**2.364**	**2.724**	**3.686**	**4.648**

TABLE X
QUANTITATIVE RESULTS (%) FOR LANGUAGE CLASSIFICATION TASKS USING REAL-WORLD DATASETS

Dataset	**Model**	$\boldsymbol{F1}_\rho$	$\boldsymbol{F1}_\omega$	$\boldsymbol{A}$
PCL [64]	LSTM	11.34	9.74	14.29
	Bi-LSTM	96.18	95.14	95.23
	2-layered Bi-LSTM	100	100	100
MPC [65]	LSTM	6.45	6.67	19.26
	Bi-LSTM	97.02	96.75	96.76
	2-layered Bi-LSTM	97.47	97.24	97.25
	3-layered Bi-LSTM	97.78	97.57	97.57

trainable parameters of the program codes for the model training, which allows it to learn a deeper code context (Table IX).

The 2-layered Bi-LSTM model specifically considers an average of 0.961 million more trainable parameters than the Bi-LSTM model, while the 3-layered Bi-LSTM model considers an average of 1.92 million more trainable parameters. In addition to considering a higher number of trainable data parameters, the layered structure and the propagation of the trainable data from one layer to another help the model to better understand the dependencies and correlations of the variables, functions, classes, tokens, and keywords of codes. However, the layered architecture can also be useful in other application domains, where tasks involve complex, diverse, and large datasets. In such cases, a layered Bi-LSTM model can be adopted by simply expanding the number of layers (e.g., $n = 3, 4, 5, \cdots$).

In addition, we conducted experiments using real-world datasets to showcase the scalability of the proposed n-layered Bi-LSTM model. We utilized two real-world datasets sourced from Project_CodeNet [62] for language classification tasks. First, we performed experiments with the Project_CodeNet_LangClass (PCL) dataset [63], which contains program code written in ten (10) programming languages, including 'Haskell', 'JavaScript', 'C#', 'C++', 'PHP', 'C', 'D', 'Rust', 'Java', and 'Python'. For these experiments, we set hyperparameter values as follows: $epoch = 50$, $\Upsilon =$ ReLu, $\beta = 64$, $\xi =$ none, and $\eta = 0.001$ for all models. The results of the language classification tasks are presented in Table X. Notably, the LSTM model did not yield favorable results compared to the Bi-LSTM and 2-layered Bi-LSTM models. The Bi-LSTM model achieved $\boldsymbol{F1}_\rho$, $\boldsymbol{F1}_\omega$, and $\boldsymbol{A}$ scores of 96.18%, 95.14%, and 95.23%, respectively. In contrast, the 2-layered Bi-LSTM model achieved 100% accuracy across all evaluation metrics, outperforming the performance of other models. It is worth noting that while the data size in the Project_CodeNet_LangClass dataset is not particularly large, the proposed layered Bi-LSTM model still produced exceptional results.

TABLE XI
MODEL PERFORMANCE WITH AND WITHOUT DENSE LAYER

Dataset/Model	**Dense Layer**	$\boldsymbol{F1}_\rho$	$\boldsymbol{F1}_\omega$	$\boldsymbol{A}$
Searching/2-lay. Bi-LSTM	Without	94.92	93.25	93.24
	With + $ReLU$	95.76	93.97	93.95
	With + $Sigmoid$	95.34	93.59	93.57
	With + $Tanh$	95.64	93.91	93.90

Furthermore, we conducted a similar experiment with another real-world dataset known as Mini_Project_CodeNet (MPC) [64], which comprises approximately 8,819 solution codes written in six (06) different programming languages, including 'C++', 'Java', 'Ruby', 'Go', 'Python', and 'C'. The number of solution codes in this dataset significantly exceeds that of PCL dataset. In this experiment, the Bi-LSTM model achieved $\boldsymbol{F1}_\rho$, $\boldsymbol{F1}_\omega$, and $\boldsymbol{A}$ scores of 97.02%, 96.75%, and 96.76%, respectively, as shown in Table X. Conversely, the 2-layered Bi-LSTM model obtained $\boldsymbol{F1}_\rho$, $\boldsymbol{F1}_\omega$, and $\boldsymbol{A}$ scores of 97.47%, 97.24%, and 97.25%, respectively, surpassing the performance of the Bi-LSTM model. Intriguingly, the 3-layered Bi-LSTM model achieved even better results than the 2-layered Bi-LSTM. This outcome underscores that the layered model possesses a deeper understanding of the code, enabling more accurate classification based on the programming languages used.

D. Ablation Studies

Since the proposed n-layered Bi-LSTM model comprises various components, including layers and hyperparameters, all of which significantly impact the overall model performance, it becomes crucial to assess the individual contributions of these components. In light of this context, we conducted a series of ablation tests aimed at elucidating the effects of these components on the model's performance. It's worth noting that these ablation tests are carried out using the 2-layered Bi-LSTM model on the Searching dataset. Initially, we investigated the influence of the Dropout Layer on model performance. When the dropout value is set to $\xi = 0.1$, the model achieved $\boldsymbol{F1}_\rho$, $\boldsymbol{F1}_\omega$, and $\boldsymbol{A}$ scores of 95.55%, 93.64%, and 93.62%, respectively. For $\xi =$ 0.3, the model attained $\boldsymbol{F1}_\rho$, $\boldsymbol{F1}_\omega$, and $\boldsymbol{A}$ scores of 95.00%, 93.21%, and 93.18%, respectively. However, when the Dropout Layer is entirely removed, the model yielded even better results, with $\boldsymbol{F1}_\rho$, $\boldsymbol{F1}_\omega$, and $\boldsymbol{A}$ scores of 95.76%, 93.97%, and 93.95%, respectively. These results indicate that the dropout layer in the proposed layered model has a relatively minor impact, leading us to conduct subsequent experiments without the dropout layer.

Furthermore, we conducted experiments both with and without the Dense Layer, as summarized in Table XI. With the dense layer, the 2-layered Bi-LSTM model achieved $\boldsymbol{A}$ scores of 93.95%, 93.57%, and 93.90% for the activation functions $ReLu$, $Sigmoid$, and $Tanh$, respectively. Conversely, when the dense layer is removed, the model obtained an $\boldsymbol{A}$ score of 93.24%. It is evident that the model's performance is adversely affected in the absence of the Dense Layer, indicating its significant contribution to the overall model performance.

E. Suitability for Programming Learning and Software Engineering

The proposed layered Bi-LSTM model yields significant results in classifying real-world program codes based on their algorithm or problem names. Searching and recognizing program codes from large code repositories are challenging task for programmers, especially for students. In this case, the proposed classification model can provide programmers/students with additional advantages in searching for and recognizing the desired program codes in large repositories. This model can ease programmers' programming method and improve his/her technical skills. The experimental results show that the proposed classification model classifies complex and diverse codes with a higher degree of accuracy. Furthermore, the model can be assimilated with existing programming learning platforms (e.g., OJ systems). Meanwhile, in SE, the reuse of software modules is one of the most important processes required to realizes a faster development. Accordingly, searching and recognizing software modules are the key tasks, to which the proposed model can be applied. The model can also be extended to various SE tasks, such as defect detection and code refactoring and review. In addition, the proposed model (classification of algorithms/codes) can be a foundation for many other machine learning models for coding tasks.

VII. Conclusion

In this paper, a layered Bi-LSTM model for the program code classification is proposed. The architecture and the theory of the layered Bi-LSTM model were also described. The deep architecture of the layered Bi-LSTM model can classify complex, large, and diverse program codes with a high degree of accuracy. The experimental results on seven real-world datasets, namely *Sorting*, *Searching*, *NC*, *G&T*, *Sorting + Searching*, *NC+BDS*, and *Sorting + Searching + G&T*, showed that the 2-layered Bi-LSTM model outperforms state-of-the-art models like the LSTM and the Bi-LSTM. Furthermore, the dataset diversity and complexity were increased when multiple datasets (i.e., *Sorting + Searching*, *NC+BDS*, and *Sorting + Searching + G&T*) were combined to verify the model performance. Accordingly, the 2-layered Bi-LSTM model achieves better classification results than the other models. Various hyperparameters (e.g., β, Υ, ξ, and η) were fine-tuned to achieve better results with the models. We also investigated the suitability of the proposed model in the domains of programming learning and software engineering.

In the future, the proposed model can be considered as a language model for generating correct codes against erroneous ones. In this case, the code pairs $< erroneous, correct >$ can be used for model training, validation, and evaluation. Furthermore, the model can generate the corresponding correct codes for the given erroneous ones.

References

[1] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "CODIT: Code editing with tree-based neural models," *IEEE Trans. Softw. Eng.*, vol. 48, no. 4, pp. 1385–1399, Apr. 2022.

[2] M. M. Rahman, Y. Watanobe, R. U. Kiran, T. C. Thang, and I. Paik, "Impact of practical skills on academic performance: A data-driven analysis," *IEEE Access*, vol. 9, pp. 139975–139993, 2021.

[3] S. Gilda, "Source code classification using neural networks," in *Proc. IEEE 14th Int. Joint Conf. Comput. Sci. Softw. Eng.*, 2017, pp. 1–6.

[4] H. Liu, M. Shen, J. Zhu, N. Niu, G. Li, and L. Zhang, "Deep learning based program generation from requirements text: Are we there yet," *IEEE Trans. Softw. Eng.*, vol. 48, no. 4, pp. 1268–1289, Apr. 2022.

[5] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Source code retrieval for bug localization using latent dirichlet allocation," in *Proc. IEEE 15th Work. Conf. Reverse Eng.*, 2008, pp. 155–164.

[6] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: A comparative study of generic and composite text models," in *Proc. 8th Work. Conf. Mining Softw. Repositories*, 2011, pp. 43–52, doi: 10.1145/1985441.1985451.

[7] R. K. Saha, J. Lawall, S. Khurshid, and D. E. Perry, "On the effectiveness of information retrieval based bug localization for C programs," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 161–170.

[8] X. Huo, M. Li, and Z.-H. Zhou, "Learning unified features from natural and programming languages for locating buggy source code," in *Proc. Int. Joint Conf. Artif. Intell.*, 2016, pp. 1606–1612.

[9] H. A. Nguyen, T. T. Nguyen, G. Wilson, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to API usage adaptation," *ACM Sigplan Not.*, vol. 45, no. 10, pp. 302–321, Oct. 2010. [Online]. Available: https://doi.org/10.1145/1932682.1869486

[10] W. Tansey and E. Tilevich, "Annotation refactoring: Inferring upgrade transformations for legacy applications," in *Proc. 23rd ACM Sigplan Conf. Object-Oriented Program. Syst. Lang. Appl.*, 2008, pp. 295–312, doi: 10.1145/1449764.1449788.

[11] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, pp. 419–428, doi: 10.1145/2594291.2594321.

[12] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S.-C. Khoo, "Semantic patch inference," in *Proc. IEEE/ACM 27th Int. Conf. Automated Softw. Eng.*, 2012, pp. 382–385, doi: 10.1145/2351676.2351753.

[13] V. Raychev, M. Schäfer, M. Sridharan, and M. Vechev, "Refactoring with synthesis," *ACM Sigplan Not.*, vol. 48, no. 10, pp. 339–354, Oct. 2013, doi: 10.1145/2544173.2509544.

[14] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *Proc. IEEE 34th Int. Conf. Softw. Eng.*, 2012, pp. 211–221.

[15] N. Meng, L. Hua, M. Kim, and K. S. McKinley, "Does automated refactoring obviate systematic editing," in *Proc. IEEE/ACM 37th Int. Conf. Softw. Eng.*, 2015, pp. 392–402.

[16] M. W. Whalen, "High-integrity code generation for state-based formalisms," in *Proc. 22nd Int. Conf. Softw. Eng.*, 2000, pp. 725–727, doi: 10.1145/337180.337615.

[17] H. Mei and L. Zhang, "Can Big Data bring a breakthrough for software automation," *Sci. China Inf. Sci.*, vol. 61, pp. 1–3, 2018, doi: 10.1007/s11432-017-9355-3.

[18] G. O'Regan, *Concise Guide to Formal Methods: Theory, Fundamentals and Industry Applications*, 1st ed. New York, NY, USA: Springer, 2017.

[19] M. M. Rahman, Y. Watanobe, and K. Nakamura, "A neural network based intelligent support model for program code completion," *Sci. Program.*, vol. 2020, pp. 1–18, 2020, doi: 10.1155/2020/7426461.

[20] R. Md Mostafizer, Y. Watanobe, and K. Nakamura, "Source code assessment and classification based on estimated error probability using attentive LSTM language model and its application in programming education," *Appl. Sci.*, vol. 10, no. 8, 2020, Art. no. 2973, doi: 10.3390/app10082973.

[21] M. M. Rahman, Y. Watanobe, and K. Nakamura, "A bidirectional LSTM language model for code evaluation and repair," *Symmetry*, vol. 13, no. 2, 2021, Art. no. 247, doi: 10.3390/sym13020247.

[22] J. Reyes, D. Ramírez, and J. Paciello, "Automatic classification of source code archives by programming language: A deep learning approach," in *Proc. IEEE Int. Conf. Comput. Sci. Comput. Intell.*, 2016, pp. 514–519.

[23] G. Fan, X. Diao, H. Yu, K. Yang, L. Chen, and A. Vitiello, "Software defect prediction via attention-based recurrent neural network," *Sci. Program.*, vol. 2019, pp. 1–14, 2019, doi: 10.1155/2019/6230953.

[24] M. Shalaby, T. Mehrez, A. El Mougy, K. Abdulnasser, and A. Al-Safty, "Automatic algorithm recognition of source-code using machine learning," in *Proc. IEEE 16th Int. Conf. Mach. Learn. Appl.*, 2017, pp. 170–177.

[25] A. Taherkhani, "Recognizing sorting algorithms with the C4.5 decision tree classifier," in *Proc. IEEE 18th Int. Conf. Prog. Comprehension*, 2010, pp. 72–75.

[26] A. LeClair, Z. Eberhart, and C. McMillan, "Adapting neural text classification for improved software categorization," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2018, pp. 461–472.

[27] X. Huo, F. Thung, M. Li, D. Lo, and S.-T. Shi, "Deep transfer bug localization," *IEEE Trans. Softw. Eng.*, vol. 47, no. 7, pp. 1368–1380, Jul. 2021.

[28] Y. Zhu et al., "Deep subdomain adaptation network for image classification," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 4, pp. 1713–1722, Apr. 2021.
[29] S. Mei, X. Li, X. Liu, H. Cai, and Q. Du, "Hyperspectral image classification using attention-based bidirectional long short-term memory network," *IEEE Trans. Geosci. Remote Sens.*, vol. 60, pp. 1–12, 2022.
[30] H. Gao, J. Xiao, Y. Yin, T. Liu, and J. Shi, "A mutually supervised graph attention network for few-shot segmentation: The perspective of fully utilizing limited samples," *IEEE Trans. Neural Netw. Learn. Syst.*, early access, Mar. 14, 2022, doi: 10.1109/TNNLS.2022.3155486.
[31] Q. Xie, P. Zhang, B. Yu, and J. Choi, "Semisupervised training of deep generative models for high-dimensional anomaly detection," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 33, no. 6, pp. 2444–2453, Jun. 2022.
[32] T. Ergen and S. S. Kozat, "Unsupervised anomaly detection with LSTM neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 31, no. 8, pp. 3127–3141, Aug. 2020.
[33] H. Gao, K. Xu, M. Cao, J. Xiao, Q. Xu, and Y. Yin, "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. Computat. Social Syst.*, vol. 9, no. 1, pp. 336–347, Feb. 2022.
[34] T. Ergen and S. S. Kozat, "Efficient online learning algorithms based on lstm neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 8, pp. 3772–3783, Aug. 2018.
[35] K. H. Dam, T. Tran, and T. Pham, "A deep language model for software code," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Soft. Eng.*, 2016, pp. 1–4. [Online]. Available: https://doi.org/10.48550/arXiv.1608.02715
[36] Y. Watanobe, M. M. Rahman, R. Kabir, and M. F. I. Amin, "Identifying algorithm in program code based on structural features using CNN classification model," *Appl. Intell.*, vol. 53, no. 10 , pp. 12210–12236, 2022.
[37] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "Sk_p: A neural program corrector for moocs," in *Proc. Companion ACM SIGPLAN Int. Conf. Syst., Program., Lang. Appl.: Softw. Humanity*2016, pp. 39–40, doi: 10.1145/2984043.29892222857.
[38] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proc. Empirical Methods Natural Lang. Process.*, 2014, pp. 1532–1543. [Online]. Available: http://www.aclweb.org/anthology/D14-1162
[39] D. Specht, "A general regression neural network," *IEEE Trans. Neural Netw.*, vol. 2, no. 6, pp. 568–576, Nov. 1991.
[40] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen netzen," *Diploma, Technische Universität München*, vol. 91, no. 1, pp. 1–74.
[41] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Trans. Neural Netw.*, vol. 5, no. 2, pp. 157–166, Mar. 1994.
[42] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997, doi: 10.1162/neco.1997.9.8.1735.
[43] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with LSTM," in *Proc. 9th Int. Conf. Artif. Neural Netw.*, 1999, pp. 850–855 .
[44] M. M. Rahman, Y. Watanobe, R. U. Kiran, and R. Kabir, "A stacked bidirectional LSTM model for classifying source codes built in MPLs," in *Proc. Mach. Learn. Princ. Pract. Knowl. Discov. Databases*, 2021, pp. 75–89.
[45] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Trans. Signal Process.*, vol. 45, no. 11, pp. 2673–2681, Nov. 1997.
[46] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *Proc. IEEE 10th Work. Conf. Mining Softw. Repositories*, 2013, pp. 207–216.
[47] A. Onan, "Two-stage topic extraction model for bibliometric data analysis based on word embeddings and clustering," *IEEE Access*, vol. 7, pp. 145614–145633, 2019.
[48] E. Sulis, D. Irazú Hernández Farías, P. Rosso, V. Patti, and G. Ruffo, "Figurative messages and affect in twitter: Differences between #irony, #sarcasm and #not," *Knowl.-Based Syst.*, vol. 108, pp. 132–143, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950705116301320
[49] Z. Hameed and B. Garcia-Zapirain, "Sentiment classification using a single-layered BiLSTM model," *IEEE Access*, vol. 8, pp. 73992–74001, 2020.
[50] C. Wang, H. Yang, and C. Meinel, "Image captioning with deep bidirectional LSTMs and multi-task learning," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 14, no. 2, pp. 1–20, Apr. 2018, doi: 10.1145/3115432.
[51] T. Liu, S. Yu, B. Xu, and H. Yin, "Recurrent networks with attention and convolutional networks for sentence representation and classification," *Appl. Intell.*, vol. 48, no. 10, pp. 3797–3806, Oct. 2018, doi: 10.1007/s10489-018-1176-4.
[52] Y. Bengio, "Learning deep architectures for AI," *Foundations Trends Mach. Learn.*, vol. 2, no. 1, pp. 1–127, Jan. 2009, doi: 10.1561/2200000006.
[53] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.
[54] Z. Cui, R. Ke, Z. Pu, and Y. Wang, "Stacked bidirectional and unidirectional LSTM recurrent neural network for forecasting network-wide traffic state with missing values," *Transp. Res. Part C: Emerg. Technol.*, vol. 118, 2020, Art. no. 102674, doi: 10.1016/j.trc.2020.102674.
[55] Y. Watanobe, "Aizu online judge," 2018. [Online]. available: https://onlinejudge.u-aizu.ac.jp/
[56] Y. Watanobe, M. M. Rahman, T. Matsumoto, U. K. Rage, and P. Ravikumar, "Online judge system: Requirements, architecture, and experiences," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 32, no. 06, pp. 917–946, 2022.
[57] R. Puri et al., "CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks," in *Proc. Neural Inf. Process. Syst. Track Datasets Benchmarks*, J. Vanschoren and S. Yeung, Eds., Curran, vol. 1, 2021, pp. 1–13. [Online]. Available: https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/a5bfc9e07964f8dddeb95fc584cd965d-Paper-round2.pdf
[58] Y. Li et al., "Competition-level code generation with alphacode," *Science*, vol. 378, pp. 1092–1097, 2022. [Online]. Available: https://arxiv.org/abs/2203.07814
[59] F. Younas, M. Usman, and W. Q. Yan, "A deep ensemble learning method for colorectal polyp classification with optimized network parameters," *Appl. Intell.*, vol. 53, no. 2, pp. 2410–2433, 2022.
[60] A. P. Bradley, "The use of the area under the ROC curve in the evaluation of machine learning algorithms," *Pattern Recognit.*, vol. 30, no. 7, pp. 1145–1159, 1997.
[61] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. 3rd Int. Conf. Learn. Representations*, 2015, pp. 1–15. [Online]. Available: http://arxiv.org/abs/1412.6980
[62] "Codenet dataset," 2021. [Online]. available: https://developer.ibm.com/exchanges/data/all/projectcodenet
[63] "Project_codenet_langclass dataset," 2021. [Online]. available: https://dax-cdn.cdn.appdomain.cloud/dax-project-codenet/1.0.0/Project_CodeNet_LangClass.tar.gz
[64] "Mini_project_codenet dataset," 2021. [Online]. available: https://dax-cdn.cdn.appdomain.cloud/dax-project-codenet/1.0.0/Mini_Project_CodeNet.tar.gz

Md. Mostafizer Rahman received the B.Sc. degree from the Department of Computer Science and Engineering, Hajee Mohammad Danesh Science & Technology University, Dinajpur, Bangladesh, in 2009, the M.Sc. engineering degree from the Dhaka University of Engineering & Technology, Gazipur, Bangladesh, in 2014, and the Ph.D. degree from the Department of Computer and Information Systems, University of Aizu, Aizuwakamatsu, Japan, in 2022. He is currently with the Dhaka University of Engineering & Technology. His research interests include machine learning, large language model, AI for Code, software engineering, NLP, big data analytics, data mining, and information visualization.

Yutaka Watanobe (Member, IEEE) received the M.S. and Ph.D. degrees from the University of Aizu, Aizuwakamatsu, Japan, in 2004 and 2007, respectively. He is currently a Senior Associate Professor with the School of Computer Science and Engineering, University of Aizu. His research interests include intelligent software, programming environment, smart learning, machine learning, data mining, cloud robotics, and visual languages. He was a Research Fellow of the Japan Society for the Promotion of Science, University of Aizu in 2007. He is currently the Director of i-SOMET. He was a Coach of four ICPC World Final teams. He is a Developer of Aizu Online Judge System. He is a Member of IEEE, IPSJ.