Hindawi
Scientific Programming
Volume 2020, Article ID 7426461, 18 pages
https://doi.org/10.1155/2020/7426461

Research Article

A Neural Network Based Intelligent Support Model for Program Code Completion

Md. Mostafizer Rahman, Yutaka Watanobe, and Keita Nakamura

School of Computer Science and Engineering, Graduate Department of Computer and Information Systems, The University of Aizu, Aizu-Wakamatsu, Fukushima 965-8580, Japan

Correspondence should be addressed to Md. Mostafizer Rahman; mostafiz26@gmail.com and Yutaka Watanobe; yutaka@u-aizu.ac.jp

Received 9 January 2020; Revised 20 May 2020; Accepted 11 June 2020; Published 14 July 2020

Academic Editor: Miroslav Bures

In recent years, millions of source codes are generated in different languages on a daily basis all over the world. A deep neural network-based intelligent support model for source code completion would be a great advantage in software engineering and programming education fields. Vast numbers of syntax, logical, and other critical errors that cannot be detected by normal compilers continue to exist in source codes, and the development of an intelligent evaluation methodology that does not rely on manual compilation has become essential. Even experienced programmers often find it necessary to analyze an entire program in order to find a single error and are thus being forced to waste valuable time debugging their source codes. With this point in mind, we proposed an intelligent model that is based on long short-term memory (LSTM) and combined it with an attention mechanism for source code completion. Thus, the proposed model can detect source code errors with locations and then predict the correct words. In addition, the proposed model can classify the source codes as to whether they are erroneous or not. We trained our proposed model using the source code and then evaluated the performance. All of the data used in our experiments were extracted from Aizu Online Judge (AOJ) system. The experimental results obtained show that the accuracy in terms of error detection and prediction of our proposed model approximately is 62% and source code classification accuracy is approximately 96% which outperformed a standard LSTM and other state-of-the-art models. Moreover, in comparison to state-of-the-art models, our proposed model achieved an interesting level of success in terms of error detection, prediction, and classification when applied to long source code sequences. Overall, these experimental results indicate the usefulness of our proposed model in software engineering and programming education arena.

1. Introduction

Programming is one of mankind's most creative and effective endeavors, and vast numbers of studies have been dedicated to improving the modeling and understanding of software code [1]. The outcomes of many such studies are now supporting a wide variety of core software engineering (SE) purposes, such as error detection, error prediction, error location identification, snippet suggestions, code patch generation, developer modeling, and source code classification [1]. Since learners and professionals around the world are constantly creating large numbers of new programs to improve our lives, it is a general truism that no program is ever released without undergoing a comprehensive postdevelopment debugging process. Almost every software product goes through different testing phases in the SE cycle. In fact, once errors are detected in the source code at the production or testing phases, the debugging process begins immediately to find and fix the errors. This means that learners and professionals are spending vast amounts of time attempting to find errors in source codes. To find a single error, it is often necessary to verify an entire program, which is a very lengthy process and time-consuming. This adverse situation has resulted in the emergence of a new SE research window [2]. There are significant numbers of errors that are commonly made by students, programmers, and developers. These include missing semicolons, delimiters, irrelevant symbols, variables, missing braces, incomplete parentheses, operators, missing methods, classes, inappropriate classes, inappropriate methods, irrelevant

parameters, logic errors, and other critical errors. Although such errors often indicate inexperience, insufficient concentration to detail, and other unsuitable behaviors, a Google study on programming showed that such errors can creep into the works of even the most experienced programmers [3]. Normally, programming is a very sensitive and error-prone task and a single mistake can eventually be harmful to software end-users. Furthermore, the source code is highly error-prone during development, so the intelligent support model for code completion has become an interesting research area. Among the solutions now being explored, the use of artificial intelligence (AI) offers fascinating potential for solving source code-related complications. In the past few years, natural language processing (NLP) developers have produced some extraordinary outcomes in different domains such as language processing, machine translation, and speech recognition. The reasons for the wide-ranging success of NLP are found in its corpus-based methods, statistical applications, messenger suggestions, handwriting recognition, and increasing large corpora of text. For example, *n*-gram models are among the stochastic language model forms that can be used for predicting the next item based on the corpus. Different *n*-gram models such as bigram, trigram, skip-gram [4], and GloVe [5] are all statistical language models that are very useful in language modeling tasks. This burgeoning usage has stimulated the availability of a large text corpus and is helping NLP techniques to become more effective on a day-by-day basis. The NLP language model is not particularly effective when used in complex SE endeavors but still useful for the intuitive language model. As a result, numerous researchers have focused their efforts on source code completion tasks using neural network-based language models. In [6], the authors proposed a local cache model that dealt with localness of software code but still encountered problems with small-context source code using an n-gram model. That study determined that neural network-based language models could provide robust substitutes for source code illustrations. Additionally, another study showed that the recurrent neural network (RNN) model, which is capable of retaining longer source code context than traditional n-gram and other language models, has achieved mentionable success in language modeling [7]. However, the RNN model faces limitations when it comes to representing the context of longer source codes due to gradient vanishing or exploding [8], which makes it hard to be trained using long dependent source code sequences. As a result, it is only effective for a small corpus of source codes. To minimize gradient vanishing or exploding problems, the RNN model has been extended to create long short-term memory (LSTM) networks [8]. An LSTM network is a special kind of RNN that can remember longer source code sequences due to its extraordinary internal structure.

In this paper, we are presenting an intelligent support model for source code completion that was designed using an LSTM in combination with an attention mechanism (then known as LSTM-AM) which increases the performances than a standard LSTM model. The attention mechanism is a useful technique that takes into account the results of all past hidden states for prediction. The attention mechanism can improve the accuracy of neural network-based intelligent models. We trained LSTM, RNN, CNN, and LSTM-AM networks with different hidden units (neurons) such as 50, 100, 200, and 300 using a bunch of source codes taken from an AOJ system. Erroneous source codes were then inputted into all models to determine their relative capabilities in regard to predicting and detecting source code errors. The obtained results show that our LSTM-AM network extends the capabilities of the standard LSTM network in terms of detecting and predicting such errors correctly. Even some source codes contain logical errors and other critical errors that cannot be detected by the usual compilers whereas our proposed intelligent support model can detect these errors. Additionally, the LSTM-AM network can retain a longer sequence of source code inputs and thus generate more accurate output than the standard LSTM and other state-of-the-art networks. In addition, we diversified with different settings and hidden units to create the most suitable model for our research in terms of cross-entropy, training time, accuracy, and other performance measurements. Also, the proposed model can classify the source codes based on the defects in codes. We expect that our proposed model will be useful for students, programmers, developers, and professionals as well as other persons involved in overall programming education and other aspects of SE.

The main contributions of this study are described as follows:

(i) The proposed intelligent support model can help students and programmers for the source code completion

(ii) The intelligent support model detects such errors (logical) that cannot be identified using conventional compilers

(iii) The proposed intelligent support model accuracy is approximately 62% which outperformed other benchmark models

(iv) Our proposed model can classify the source codes based on the detected errors. The classification accuracy is 96% which is much better than other models

(v) The proposed model highlights defective spots with location/line number in source codes

(vi) The proposed model improves the ability of learners to fix errors in source code easily by using the location/line numbers

The remainder of the paper is structured as follows. In Section 2, we present the background of our study and discuss prior research. Section 3 describes the overview of natural language processing and artificial neural networks. In Section 4, we present our proposed approach. Data collection and problem description issues are presented in Section 5. The experimental results are described in Section 6. In Section 7, we discuss the results. To that end, Section 8 concludes this paper with some future work proposals.

2. Background and Prior Research

Modern society is flourishing due to advancements in the wide-ranging fields of information and communication technology (ICT), where programming is a crucial aspect of many developments. Millions of source codes are being created every day, most of which are tested through manual compiling processes. As a result, an important research field that has recently emerged involves the use of AI systems for source code completion during development rather than manual compiling processes. More specifically, artificial neural network-based models are using for source code completion in order to achieve more humanlike results. Numerous studies have been completed and a wide variety of different methods were proposed regarding the use of AI in SE fields, some of which will be reviewed below.

In [1], the authors present a language model for source code testing that uses a neural network instead of an existing language (i.e., *n*-gram) model. In most cases, *n*-gram language models cannot handle long source code sequences effectively, so neural network-based language models were developed to improve source code analyses. In the cited study, RNN and LSTM language models were trained and the obtained results showed that the LSTM model performed better than the RNN model. That study used a Java project corpus to evaluate the performance of the language models.

In [4], the authors proposed a novel LSTM-based source code correction method that used segment similarities. More specifically, the study utilized the sequence-to-sequence (seq2seq) model for the source code correction process. The seq2seq model is a machine learning approach which is very effective for language modelings such as machine translator, conversational model, text translator, and image captioning.

White et al. [7] proposed a deep software language model using RNN. The obtained experimental results using a Java corpus showed that the proposed software language model outperformed conventional models like cache-based *n*-gram and standard n-gram. That software language model showed significant promise for use in SE fields.

In [9], the authors presented a novel Tree-LSTM-based model where each LSTM unit was used as a tree. That model assesses semantic relatedness prediction tasks based on sentence pairs and sentiment classification. Meanwhile, in [10], the authors proposed a method that classified archived source codes by language type using an LSTM. Their experimental results demonstrated that the proposed LSTM surpassed the linguist classifier, Naive Bayes (NB) classifier, and other similar networks.

In [11], the authors proposed a technique that automatically identifies and corrects source code syntax errors using an RNN. Their proposed SYNFix algorithm finds the error location of the next predicted token sequence, after which the identified error is solved by either replacing or inserting a proper word. A significant limitation of this technique is that it cannot recover or handle multiple syntax errors in a source code sequence.

Pedroni and Meyer [12] studied to find the appropriate type of compiler messages that can assist novice programmers to identify source code errors. And what actions are needed for the error messages? In that study, the authors experimentally showed that certain message types help novice programmers more than others.

In [13], the authors presented a model for source code syntax error correction that was written in the C programming language. That model, called DeepFix, uses a multilayered sequence-to-sequence neural network combined with an attention mechanism. The authors also proposed a trained RNN that can predict an error with its location number and fix the error with a proper token. The experimental results obtained showed that, out of a total of 6971 source code errors, this approach completely fixed about 27% and partially fixed about 19%.

Rahman et al. [14] proposed a language model using LSTM for fixing source code errors. The proposed model is a combined attention mechanism with LSTM which increases the effectiveness of standard LSTM. Experimental results showed that the model significantly corrected errors in the source codes.

In [15], the authors proposed a source code bug detection technique that works by varying the hyperparameters of an LSTM in order to investigate perplexity issues and training time. The results obtained show that LSTM produces significant results for source code error detection.

Bahdanau et al. [16] proposed a language translation model that uses RNN. More specifically, the encoder-decoder technique is used as a translator when it is necessary to encode a source text into a fixed-length vector. By utilizing the vector length, the decoder can translate the sentences. The paper extends fixed-length limitations by allowing (soft) search from the source sentence to predict a target word instead of using the hard segments of the source code sentences.

Li et al. [17] points out the limitations of neural network language models. To overcome those problems, the authors proposed two new approaches: an attention mechanism-enhanced LSTM and a pointer mixture network. The attention mechanism-enhanced LSTM is used to alleviate fixed-size vectors and improve memorization capability by providing a variety of ways for gradients to backpropagate. In contrast, the pointer mixture network predicts out of vocabulary (UoV) words by considering locally repeating tokens. That study also proposed the use of an abstract syntax tree- (AST-) based code completion method.

Li et al. [18] presented a source code defect prediction model using a convolution neural network called DP-CNN. Abstract syntax tree (AST) has used to convert the source code into token vectors. Using the word embedding map, each token vector is converted into a numerical vector. CNN used a numerical vector for training. Thereafter, the CNN model creates the source code's semantic and structural features. Compared with the traditional defect prediction features, DP-CNN improved the performance by 12% compared with other state-of-the-art models and 16% compared with other traditional feature basis methods.

Dam et al. [19] presented a deep learning model for software defect prediction. The model has used the abstract syntax tree (AST) incorporated in the LSTM network. Each node of the AST structure is treated as an LSTM unit. A deep tree-based LSTM model stored syntactical and structural information of source codes for accurate prediction. The learning style of the tree-based LSTM model is unsupervised. The model does not clean or replace any erroneous words by predicting correct words. It is used to generate error probability from a source code; thereafter; a classifier identifies the source code's defect by using the value of probability.

Pham et al. [20] used CNN as a language model based on Feed Forward Neural Network (FFNN). Experimental results demonstrated that the performance of the CNN language model is better than the normal FFNN. As for recurrent models, the CNN language model performs well compared with the RNN, but below the state-of-the-art LSTM model.

In [21], the authors proposed an RNN-based model for the source code fault prediction. There are two familiar evaluation methods such as the area under the curve (AUC) and F1-measures that are used to measure the performance. The proposed attention-based RNN model improves the accuracy of source code classification. The AUC and F1-measure achieved 7% and 14% more accuracy than the other benchmark models.

In summary, a wide variety of methods and techniques have been proposed in various studies, most of which used RNN, LSTM, or convolutional neural network (CNN) models for source code manipulation and other applications. It is very difficult to explain which proposed research work is superior over other researches. RNNs perform comparatively better than conventional language models, but RNNs have limited ability to handle long source code inputs [7]. An LSTM is a special kind of RNN network that can remember longer source code sequences due to its extraordinary internal structure and thus overcome RNN shortcomings. Our proposed model is unlike the other models. Our proposed LSTM-AM network further extends the capabilities of a standard LSTM network to the point where it can be used for detecting and predicting source code errors as well as source code classification. Standard LSTM network only uses the last hidden state to make predictions. In contrast, our LSTM-AM network can take the outcomes of all previous hidden states into consideration when making predictions. Therefore, it is a more promising technique for use in source code manipulations than other state-of-the-art language models.

3. Overview of Language Model and Artificial Neural Networks

3.1. N-Gram Language Model. The resources of natural text corpora are being enriched by the accumulation of text from multiple sources on a daily basis. The success behind natural language processing is based on this rich text corpus. For this reason, and because of their simplicity and scalability, n-gram models are popular in the field of natural language processing. An n-gram model predicts the upcoming word or text of a sequence based on probability, and the probability of an entire word sequence $P(w_1, w_2, \ldots, w_n)$ can be calculated by using the chain rule of probability.

$$P\left(W_1^n\right) = P(W_1)P(W_2 \mid W_1)P\left(W_3 \mid W_1^2\right)\ldots P\left(W_n \mid W_1^{n-1}\right) = \prod_{i=1}^{m} W_i \mid W_1^{i-1}. \tag{1}$$

The Markov assumption, which is used when the probability of a word depends solely upon the previous word, is described in

$$P\left(W_n \mid W_1^{n-1}\right) \approx P(W_n \mid W_{n-1}). \tag{2}$$

Thus, the general equation of an n-gram used for the conditional probability of the next word sequence is as follows:

$$\prod_{i=1}^{m} W_i \mid W_1^{i-1} \approx \prod_{i=1}^{m} W_i \mid W_{i-n+1}^{i-1}. \tag{3}$$

In practice, the maximum likelihood can be estimated by many smoothing techniques [22], as shown in the following equation:

$$P\left(W_i \mid W_{i-n+1}^{i-1}\right) = \frac{C\left(W_{i-n+1}^{i-1} W_i\right)}{C\left(W_{i-n+1}^{i-1}\right)}. \tag{4}$$

Cross-entropy is measured to validate the prediction goodness of a language model [23]. Low cross-entropy values imply better language models.

$$H_p \approx -\frac{1}{m}\sum_{i}^{m} \log_2 p\left(W_i \mid W_{i-n+1}^{i-1}\right). \tag{5}$$

3.2. Recurrent Neural Network (RNN). An RNN is a neural network variant that is frequently used in natural language processing, classification, regression, etc. In a traditional neural network, inputs are processed through multiple hidden layers and then output via the output layer. In the case of sequential dependent input, a general neural network cannot manufacture accurate results. For example, in the case of the dependent sentence "*Rose is a beautiful flower*," a general neural network takes the "*Rose*" input to produce an output based solely on "*Rose*." Then, when the word "*is*" input is considered, the network does not use the previous of "*Rose*" result. Instead, it simply produces the result using the word "*is*." Similarly, a simple neural network takes other words "*a*," "*beautiful*," and "*flower*" to generate results without considering the previous result of inputs. To address this problem, RNN has emerged with an internal memory that retains previous time step results. A simple RNN structure is shown in Figure 1.

Mathematically, an RNN can be presented using equation (6). The current state of the RNN can be expressed as

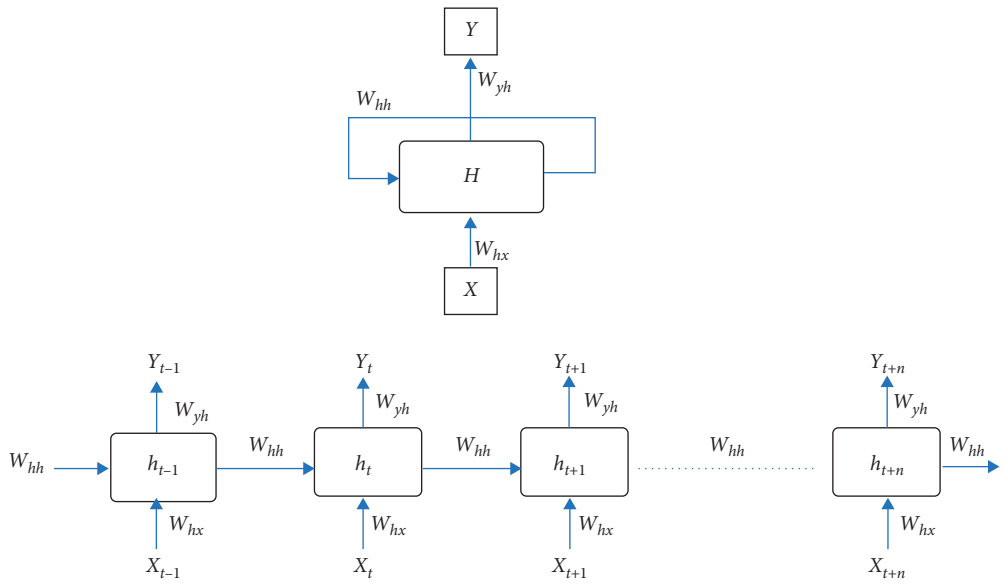


Figure 1: A simple RNN structure.

$$h_t = f\left(h_{t-1}, x_t\right), \tag{6}$$

$$h_t = f\left(w_{hh}h_{t-1} + w_{hx}x_t\right), \tag{7}$$

where h_t is the current state, h_{t-1} is the previous state, x_t is the current state input, w_{hh} is the weight of the recurrent neuron, and w_{hx} is the weight of the input neuron.

Equation (8) is used as an activation function (tanh) of RNN:

$$h_t = \tanh\left(w_{hh}h_{t-1} + w_{hx}x_t\right). \tag{8}$$

Finally, the output function can be written as follows:

$$y_t = \text{softmax}\left(w_{yh}h_t\right), \tag{9}$$

where w_{yh} is the weight for the output layer and y_t is the output.

RNN has multiple input and output types such as one to one, one to many, many to one, and many to many. Despite all the advantages, RNN is susceptible to the major drawbacks of gradient vanishing or exploding.

3.2.1. Gradient Vanishing and Exploding. In this section, we discuss the RNN gradient vanishing and exploding problems. It seems simple to train the RNN network, but it is very hard because of its recurrent connection. In the case of forward propagation, we multiply all the weight metrics and a similar procedure needs to apply the backpropagation. For the backpropagation, the signal may be strong or weak which causes exploding and vanishing. Gradient vanishing makes a complex situation to determine the direction of model parameters to improve the loss function. On the other hand, exploding gradients make the learning condition unstable. Training of the hidden RNN network is passed through different time steps using backpropagation. The sum of a distinct gradient at every time step is equal to the total error gradient. The error can be expressed by considering total time steps T in the following equation:

$$\frac{\partial E}{\partial L} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial L}. \tag{10}$$

Now, we apply the chain rule to calculate the overall error gradients:

$$\frac{\partial E}{\partial L} = \sum_{t=1}^{T} \frac{\partial E}{\partial y_t}\frac{\partial y_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial L}. \tag{11}$$

The term $\partial h_t/\partial h_k$ is involved with the product of Jacobians $\partial h_i/\partial h_{i-1}$ as shown in the following equation:

$$\frac{\partial h_t}{\partial h_j} = \frac{\partial h_t}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial h_{t-2}}\frac{\partial h_{t-2}}{\partial h_{t-3}}\frac{\partial h_{t-3}}{\partial h_{t-4}}\cdots\frac{\partial h_{j+1}}{\partial h_j}, \tag{12}$$

$$= \prod_{i=j+1}^{t} \frac{\partial h_i}{\partial h_{i-1}}. \tag{13}$$

The term $\partial h_t/\partial h_{t-1}$ in equation (12) is evaluated by equation (7).

$$\prod_{i=j+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=j+1}^{t} W^T \text{diag}\left[f'\left(h_{i-1}\right)\right]. \tag{14}$$

Now, by the Eigen decomposition on the Jacobian matrix $\partial h_t/\partial h_{t-1}$ given by $W^T\text{diag}[f'(h_{i-1})]$, we obtain the eigenvalues $\lambda_1, \lambda_2, \lambda_3, \ldots, \lambda_n$ where $\lambda_1 > \lambda_2 > \lambda_3 > \cdots > \lambda_n$ and the corresponding eigenvectors are $v_1, v_2, v_3, \ldots, v_n$. If the direction of a hidden state Δh_t is moved to v_i by any modifications, then the gradient will be $\lambda_i \Delta h_t$. From equation (14) the product of the Jacobians of the hidden state sequences is $\lambda_i^1\Delta h_1, \lambda_i^2\Delta h_2, \lambda_i^3\Delta h_3, \ldots, \lambda_i^n\Delta h_n$. It is easy to visualize the term λ_i^t dominating Δh_t. In summary, if the greatest eigenvalue is $\lambda_1 < 1$, then the gradient will vanish and $\lambda_1 > 1$ causes the gradient exploding [24]. To alleviate the

gradient vanishing or exploding problems, the gradient clipping, input reversal, identity initialization, weight regularization, LSTM, etc. techniques can be used.

3.3. LSTM Network. An LSTM neural network is a special kind of RNN network that is often used to process long inputs. An LSTM is not limited to a single input but can also process complete input sequences. Usually, an LSTM is structured with four gates such as forget, input, cell state, and output. Each gate has a separate activity where the cell state keeps complete information of the input sequences and others are used to manage the input and output activities. Figure 2 shows the structure of a basic LSTM unit.

At the very beginning, processing starts with the forget gate to determine which information has to be discarded from (or retain in) the cell state. The forget gate in cell state c_{t-1} can be expressed by the following equation (15) where h_{t-1} is the hidden state and x_t is the current input. The output (0 or 1) of the forget gate is produced through a sigmoid function. If the result of the forget gate is 1, then we keep the data in the cell state; otherwise, we discard the data.

$$f_t = \sigma\left(W_f\left[h_{t-1}, x_t\right] + b_f\right). \tag{15}$$

The input gate determines which cell state value should be updated when new data appears. Through the tanh function, the candidate value $\widetilde{c}_t$ for the cell state is now created.

$$\begin{aligned} i_t &= \sigma\left(w_i \cdot \left[h_{t-1}, x_t\right] + b_i\right), \\ \widetilde{c}_t &= \tanh\left(w_c \cdot \left[h_{t-1}, x_t\right] + b_c\right). \end{aligned} \tag{16}$$

Now, we update the old cell state c_{t-1} by the c_t

$$c_t = f_t * c_{t-1} + i_t * \widetilde{c}_t. \tag{17}$$

The filtered version of the cell state will be output o_t via the sigmoid function and the weight will also be updated.

$$\begin{aligned} o_t &= \sigma\left(w_o \cdot \left[h_{t-1}, x_t\right] + b_o\right), \\ h_t &= o_t * \tanh\left(c_t\right). \end{aligned} \tag{18}$$

Recognizing the strength of LSTM, we were motivated to apply this network model to error detection, prediction, correction, and classification in source codes.

4. Proposed Approach

Our proposed LSTM-AM network has an effective deep learning architecture that is used as an intelligent support model for source code completion. Accordingly, we trained our model using correct source codes and then used it successfully to detect errors and predict correct words in erroneous source codes based on the trained corpus. Moreover, the proposed model can classify the source codes using the prediction results. Our model generates a complete feedback package for each source code after being examined where learners and professionals can benefit from the model. The workflow of our proposed model is depicted in Figure 3.

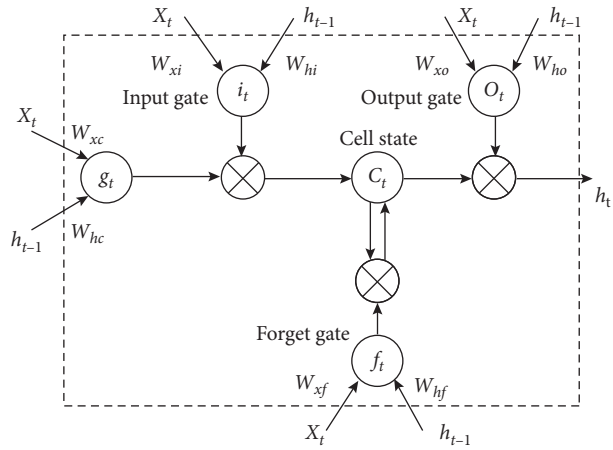


Figure 2: Internal structure of an LSTM unit.

4.1. Proposed LSTM-AM Network. Over the years, attention mechanisms have been adapted to a wide variety of diverse tasks [25–30], the most popular and effective of which is sequence-to-sequence modeling. Typically, in sequence-to-sequence modeling, the output of the last hidden state is used as the context vector for further consideration. It is very difficult to process long sequenced inputs using the sequence-to-sequence model [31]. The attention mechanism makes it possible to map all previously hidden state outputs, including the latest hidden state output, to produce the most relevant and accurate results.

With this point in mind, we incorporated an attention mechanism into a standard LSTM to make LSTM-AM, as shown in Figure 4. This strengthens our model's ability to predict longer source code sequences. Attention usually improves the performance of the language and translator model by merging all hidden state outputs with the softmax function; sometimes, attention mechanisms work as a dense layer. Recently, attention mechanisms have been used in machine translation tasks with great success. Furthermore, sometimes it is necessary for a machine translator model to compress entire input sequences into a smaller size vector, so there is a possibility of information loss. The use of attention mechanisms has fixed this problem. In our proposed model, an attention mechanism is combined with an LSTM. Although the abilities of a standard LSTM to capture long-range dependencies are far superior to those of an RNN. It still encounters problems when a hidden state has to carry all the necessary data in a small-sized vector [31]. The introduction of attention mechanisms and their alignment with neural language models such as LSTM are aimed at overcoming these problems [16]. The attention mechanism offers neural language models to bring and use appropriate information in all secret states of the past. As a result, the network's retention ability is improved and diverse paths are provided for gradients to backpropagate. More detailed mathematical illustrations of attention mechanisms can be found in [17].

For our attention mechanism, we took the external memory of E for the previous hidden states, which is denoted by $M_t = [h_{t-E} \ldots h_{t-1}] \epsilon \mathbb{R}^{k * E}$. The proposed model

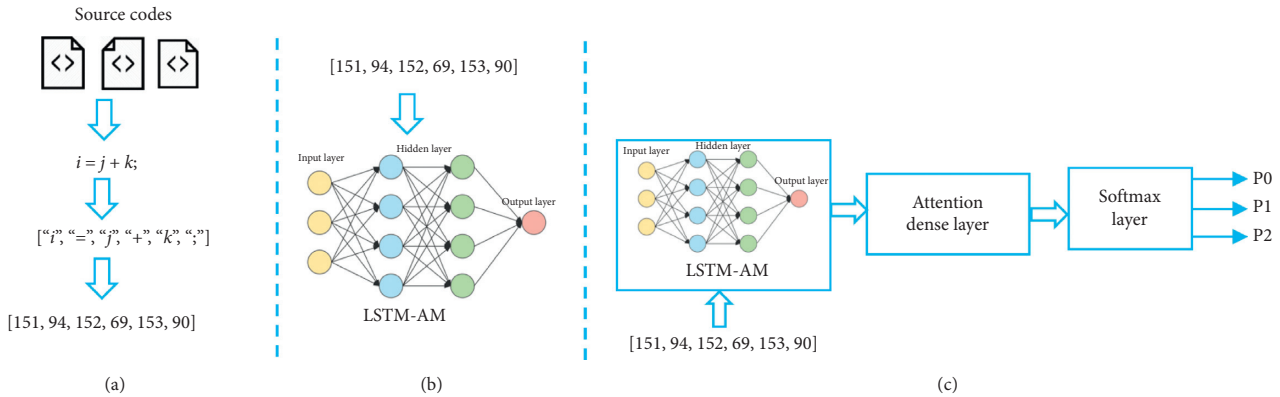


FIGURE 3: The main framework of our model: (a) conversion of source code to token IDs, (b) model training using token IDs, and (c) results produced by the softmax function.

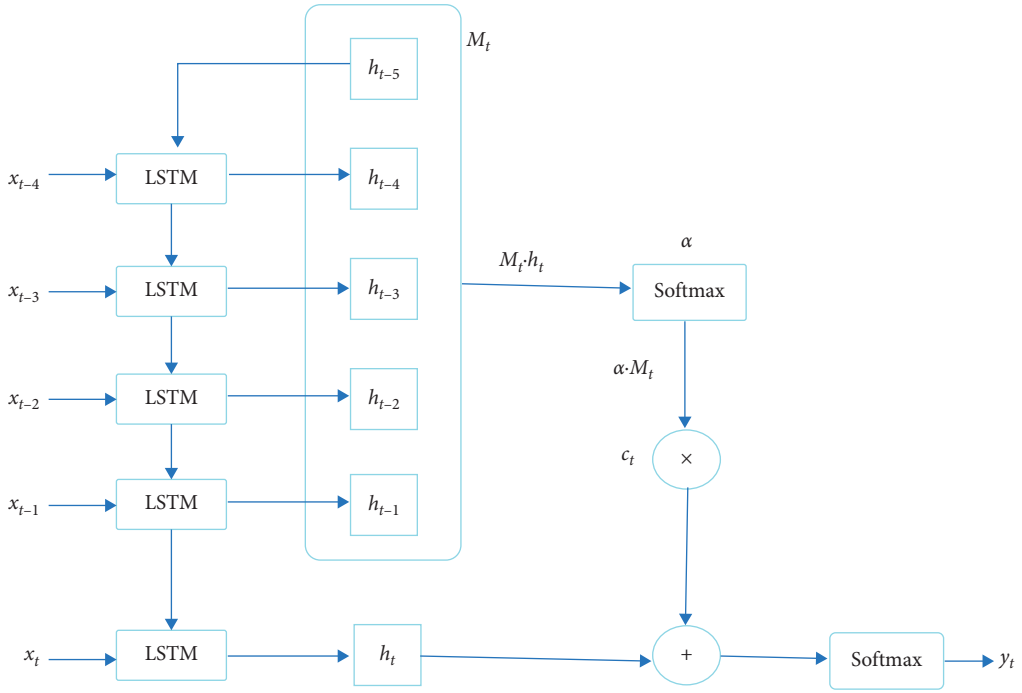


FIGURE 4: An architecture of the proposed LSTM-AM network.

used attention layer by considering h_t and M_t at the time t, attention weight α_t, and context vector c_t.

$$\begin{aligned} A_t &= M_t \cdot h_t, \\ \alpha_t &= \text{softmax}(A_t), \\ c_t &= M_t \alpha_t^T. \end{aligned} \tag{19}$$

To predict the next word at time step t, judgment is based not only on current hidden states h_t but also on context vector c_t. At that point, the focus turns to the vocabulary spaces to produce the final probability $y_t \epsilon \mathbb{R}^v$ via softmax function. Here, G_t is an output vector.

$$\begin{aligned} G_t &= \tanh\left(w^g \left[w^h(h_t) + w^m(c_t)\right]\right), \\ y_t &= \text{softmax}\left(w^v G_t + b^v\right), \end{aligned} \tag{20}$$

where $w^g \epsilon \mathbb{R}^{k*2k}$ and $w^v \epsilon \mathbb{R}^{v*k}$ are trainable projection matrices, $b^v \epsilon \mathbb{R}^v$ is a bias, and v is a vocabulary/dictionary size.

Based on the above aspects, we can see that the use of an attention mechanism helps to effectively extract the exact features from input sequences. As such, the use of LSTM-AM will increase the capability of our model.

5. Data Acquisition and Problem Description

An online judge (OJ) system is a web-based programming environment that compiles and executes submitted source codes and returns judgments based on test data sets. OJ system is an open platform for programming practice as well as competition. To conduct our experimental work, we

collected source code from the AOJ system [32, 33]. Currently, the AOJ system is effortlessly performing for various programming competitions and academies. As of May 2020, about 75,000 users are regularly playing their programming activities on the AOJ platform, with 2100 autonomous problem sets. All problem sets are classified based on different algorithms and branches of computer science [14]. As a result, about 4.5 million massive solution source codes have been archived on the AOJ platform, encouraging better research in the field of software engineering. We used all source codes from the AOJ system for training and testing purposes to avoid threats or difficulties in our proposed model. For model training, we selected all of the correct solutions written in C language of the three problems such as the greatest common divisor (GCD), insertion sort (IS), and prime numbers (PN). There are a total of 2285 correct source codes for the IS problem and the overall solution success rate is 35.16%. The total number of correct source codes for the GCD problem is 1821 and the overall solution success rate is 49.86%. Considering the GCD problem, we see that there are two inputs (a and b) given in a line, after which the greatest common divisor of a and b will be output, as shown in Figure 5(a).

In contrast, the total number of correct source codes for the PN problem is 1538 and the overall solution success rate is 30.8%. In the PN problem description, the first line contains an integer N. The code needs to count the number of prime numbers in the list of N elements, as shown in Figure 5(b).

5.1. Data Preprocessing and Training. Before we conducted training, raw source codes were filtered by removing unnecessary elements. To accomplish this, we followed the procedure applied to [14] for source code embedding and tokenization. First, we removed all irrelevant elements from codes like lines (n), comments, and tabs ($\backslash t$). After that, all the remaining elements of the code were converted into word sequences where numbers, functions, tokens, keywords, variables, classes, and characters were treated as simple words. The whole code transformation process is called tokenization and vocabulary creation. Then, each word was encoded with IDs in which the function names, keywords, variable names, and characters were encoded as listed in Table 1. The flowchart of the training and evaluation process of our model is shown in Figure 6.

At the early stage of the training phase, the source codes were first converted into word sequences and then encoded into token IDs as shown in Figure 7. This conversion process is called word embedding and tokenization.

Upon completion of the embedding and tokenization process, we trained our proposed model and other related state-of-the-art models with the correct source codes of IS, GCD, and PN problems. The simple training process of an LSTM-based language model is shown in Figure 8.

At the end of the training process, the next step is to check the performance of the model for the source code completion task. How accurately it identifies errors and predicts corrections? Our proposed model created the probability for each word. We considered a word will be an error candidate whose probability is below 0.1 [14]. Additionally, to test the model loss function, we calculated the cross-entropy for each epoch at the softmax layer. Cross-entropy is defined as the difference between actual and predicted results. Softmax is an activation function that creates probabilities. Typically, softmax is used as the last layer of neural networks. The output range of the softmax function is between 0 and 1. The softmax layer received $x = [x_1, x_2, x_3, x_4, \ldots, x_n]$ and returns probability $p = [p_1, p_2, p_3, p_4, \ldots, p_n]$, as defined in the following equation:

$$P_i = \frac{\exp(x_i)}{\sum_{j=1}^{k} \exp(x_j)}. \tag{21}$$

Cross-entropy is an effective performance measurement indicator for the probability-based model. Low-valued cross-entropy indicates a good model.

$$H_p \approx -\frac{1}{m} \sum_{i}^{m} \log_2 p\left(W_i \mid W_{i-n+1}^{i-1}\right). \tag{22}$$

A simple example of the prediction process used by our model is shown in Figure 9. An input sequence example is {"=," "x," "+," "y"}; then the model calculates the next probable correct word based on the source code corpus. Finally, the word with the highest probability is the winner of the next predicted word. Based on the input sequence in the example above, the correct predicted word is {";"}.

5.2. Hyperparameters. In the present research, we defined several experimental hyperparameters in order to obtain better results. To avoid overfitting, a dropout ratio (0.3) was used for our proposed model. The LSTM network was optimized using Adam, which is a stochastic optimization method [34, 35]. The learning rate is an important factor for neural network training because the value of the learning rate can control the learning speed of the model. Network learning becomes faster and slower on the basis of higher and lower value of learning rates, respectively. In the present paper, we determine the learning rate $l = 0.002$, and the network weights during training are updated by the value of l. β_1 is the exponential decay rate for the first-moment estimate and the second-moment estimate of the exponential decay rate is β_2. The values of the β_1 and β_2 are 0.001 and 0.999, respectively. The value of ε chosen to avoid any division by zero which is $\varepsilon = 1e^{-8}$. We trained our network in 50, 100, 150, and 200 hidden units. Each model type was named with reference to the number of units, such as the 100-unit model and 200-unit model. After training, we assessed the ability of our proposed LSTM-AM technique to pick the best number of hidden units from the created models.

6. Experimental Results

Our proposed intelligent support model can be useful for source code completion. Also, it is a general model and can be adapted to any source code for model training and testing.

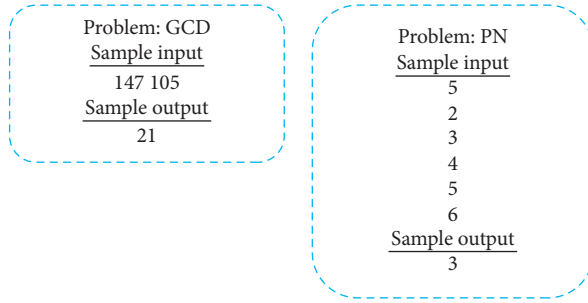


Figure 5: Input and output sample of GCD and PN problems.

Table 1: A partial list of IDs for characters, special characters, numbers, and keywords.

Words	IDs
Auto, break, char, case, const, continue, do, default, double, enum, else, exturn, for, float, goto, int, if, long, return, register, signed, short, sizeof, struct, static, switch, typedef, unsigned, union, volatile, void, while	30–45
A to Z (upper) and a to z (lower)	96–121 and 127–152
0–9	80–89
=, ∗, +, −, /, %, |, <, >, (), {, }, [,]	94, 74, 75, 76, 79, 69, 155, 92, 93, 72, 73, 153, 155, 122, 124
@, #, $, %, and	95, 67, 68, 69, 70
!(Not), ?(question), ‘(single quote), “(double quote),.(dot),; (semicolon),:(double colon),,(comma)	63, 64, 71, 66, 78, 90, 91, 76

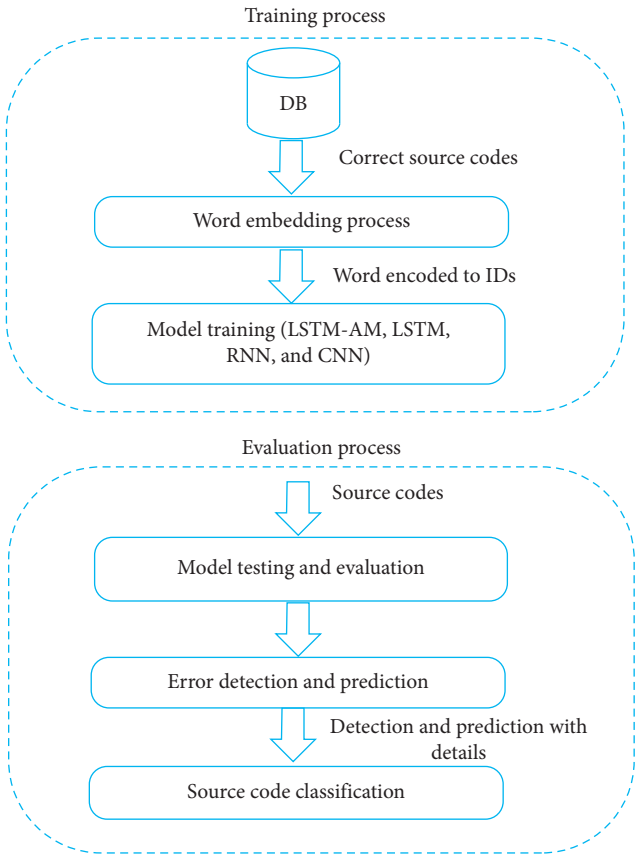


Figure 6: The flowchart of the training and evaluation process of our proposed model.

In our proposed model, we defined a minimum probability value by which the model can identify error candidate words based on the training corpus. Accordingly, we randomly chose some incorrect IS, GCD, and PN source codes and used them to evaluate the models' performance levels. Here, we should note that all of our research work and language model training were performed on an Intel® Core™ i7-5600U central processing unit (CPU) personal computer clocked at 2.60 GHz with 8 GB of RAM in a 64-bit Windows 10 operating system.

6.1. Hidden Unit Selection and Cross-Entropy Measurement. We used several hidden units such as 50, 100, 150, and 200 to train our proposed LSTM-AM and other state-of-the-art models. In training, the correct source codes of IS, GCD, and PN problems are used separately and all the source codes of IS, GCD, and PN are used combinedly. The number of source codes of each type of problem is listed in Table 2.

We trained our proposed LSTM-AM and different state-of-the-art models using correct source codes. Table 3 presents the cross-entropy in 30 epochs during training using PN source codes. The 50-, 100-, 150-, and 200-unit models took a total of 11483, 20909, 38043, and 59065 seconds to train the LSTM-AM model using the PN problem, respectively.

Tables 4 and 5 present cross-entropy of different models during training using GCD and IS source codes, respectively. The 50-, 100-, 150-, and 200-unit models took a total of 19005, 24110, 24273, and 30420 seconds to train the LSTM-AM model using the GCD problem, respectively, and for the

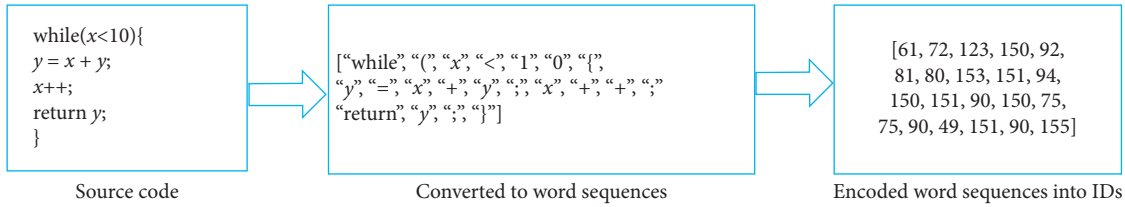


Figure 7: Source code embedding and tokenization process.

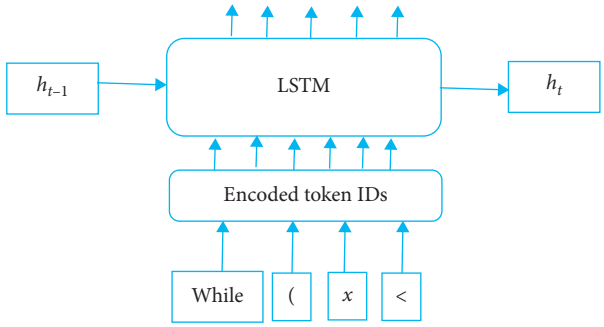


Figure 8: Training process of an LSTM language model.

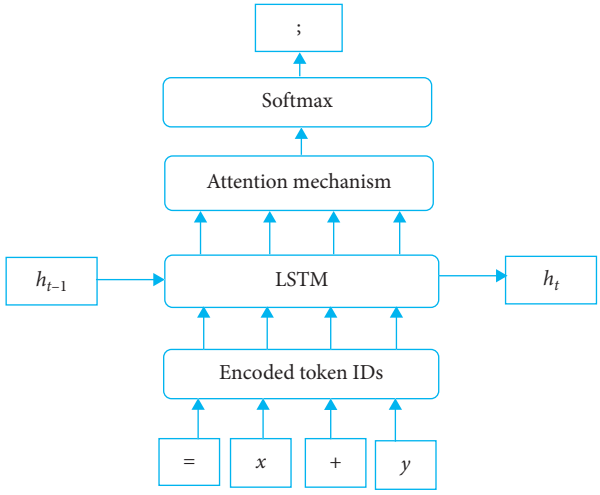


Figure 9: LSTM-AM network prediction process.

Table 2: Number of source codes of each problem.

Problem type	Number of source codes
Greatest common divisor (GCD)	964
Insertion sort (IS)	1518
Prime numbers (PN)	972

Table 3: Cross-entropy comparison using PN source codes.

Model	Units (neurons)			
	50	100	150	200
LSTM-AM	4.35	3.90	2.87	2.23
LSTM	4.75	3.31	2.37	2.02
RNN	6.35	4.72	4.21	3.95

Table 4: Cross-entropy comparison using GCD source codes.

Model	Units (neurons)			
	50	100	150	200
LSTM-AM	2.22	1.80	1.75	1.30
LSTM	2.56	1.91	1.80	1.39
RNN	5.11	4.36	3.50	3.23

Table 5: Cross-entropy comparison using IS source codes.

Model	Units (neurons)			
	50	100	150	200
LSTM-AM	3.12	1.55	1.40	1.27
LSTM	3.26	1.63	1.48	1.26
RNN	4.99	3.78	2.89	3.11

IS problem, it took a total of 39643, 62756, 80100, and 100803 seconds, respectively. In contrast, other models such as LSTM and RNN took relatively less time for training.

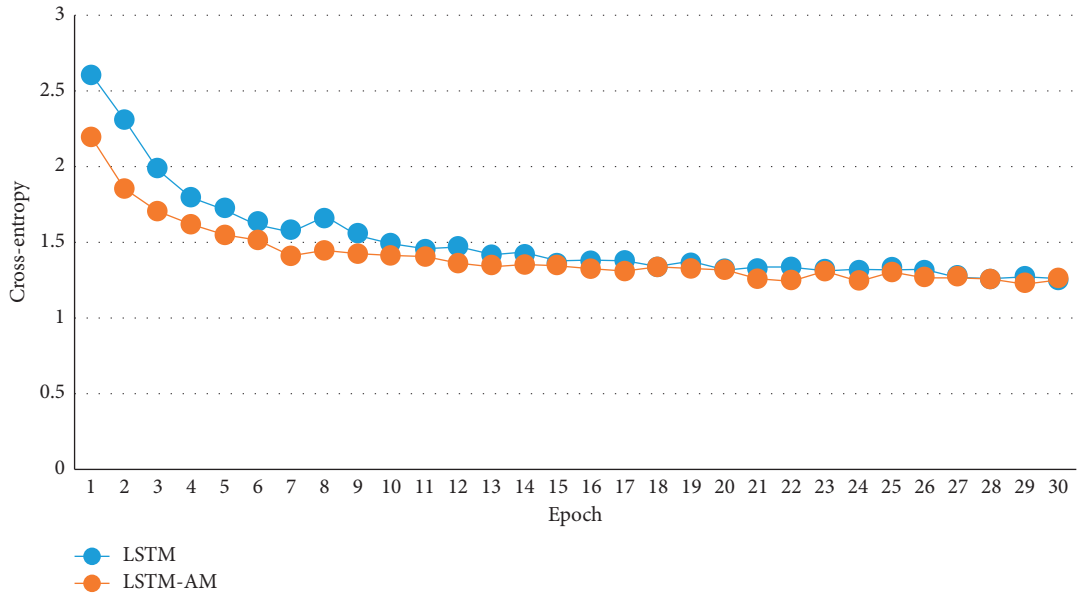
To evaluate the efficiency of the proposed model, epochwise cross-entropy during the training periods using the 200-unit model was calculated which is depicted in Figure 10.

As mentioned above, the efficiency of a model strongly depends upon the value of cross-entropy. During training, the 200-unit model produced the lowest cross-entropy using each type of problem set. The cross-entropy of the 200-unit model using IS, PN, and GCD problems is shown in Figure 11.

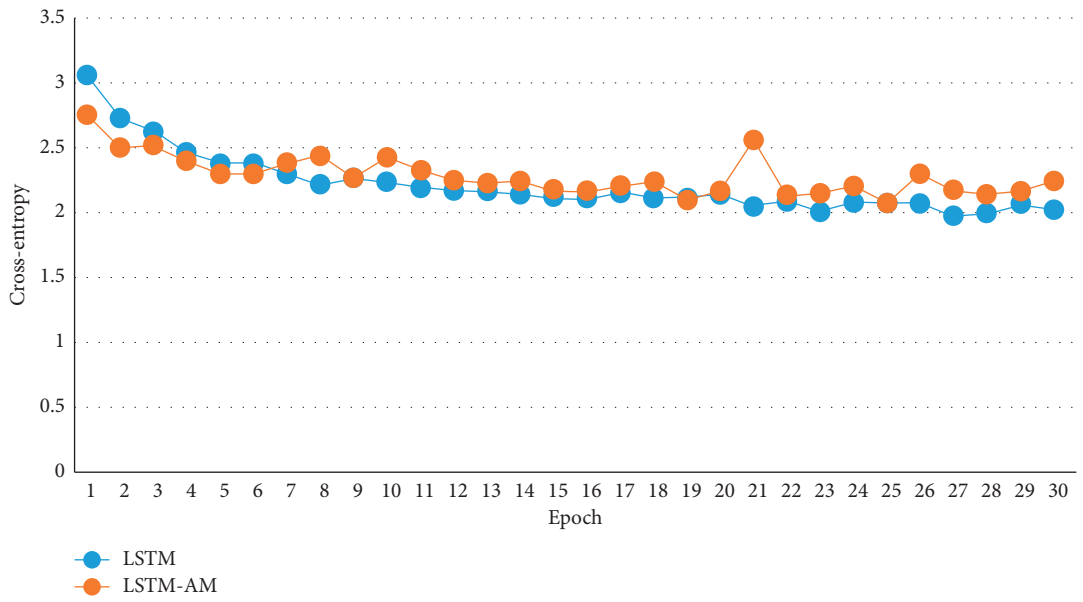
We aimed to find the best-suited hidden units for our LSTM-AM network and other state-of-the-art models. In this regard, we put together all the source codes (about 3442) to train our proposed and other state-of-the-art models. The cross-entropies and total times are recorded at the last epoch of all the models as presented in Table 6. The cross-entropy of the 200-unit model is lower than other models.

Based on the above aspect, it is ensured that the 200-unit model provides the best results because its cross-entropy is the lowest value among all the units; thus, we selected a 200-unit model for the LSTM-AM network and other state-of-the-art networks.

6.2. Error Detection and Prediction. In our evaluations, we tested LSTM-AM and other state-of-the-art models using erroneous source codes. Probable error locations were marked by changing the text color and underlining the suspected erroneous portions. Also, the proposed model

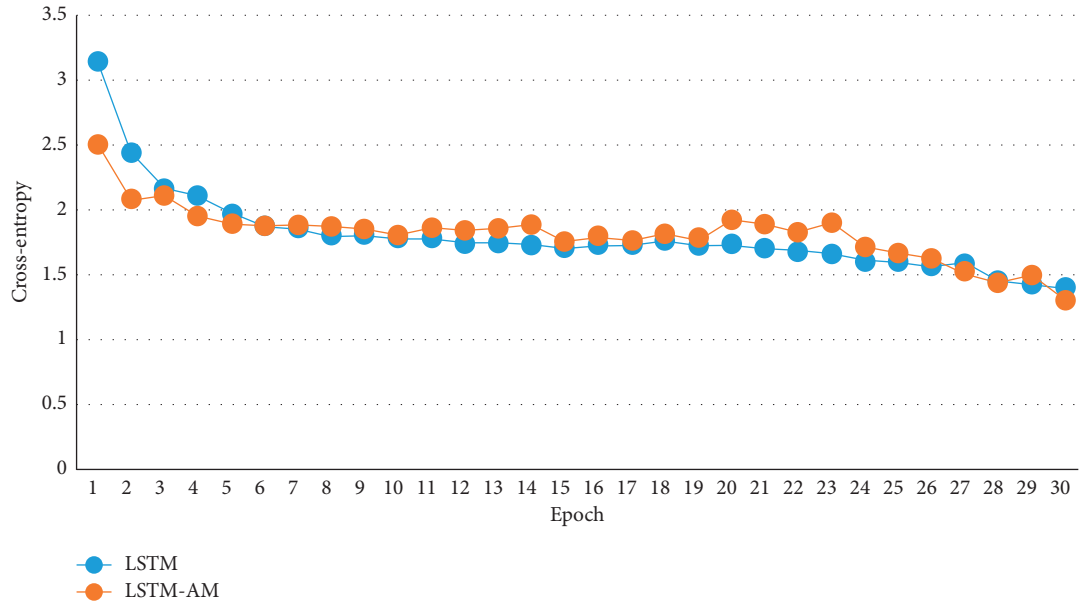


(a)



(b)

Figure 10: Continued.



(c)

Figure 10: Epochwise cross-entropies of 200-unit model using (a) IS, (b) PN, and (c) GCD source codes.

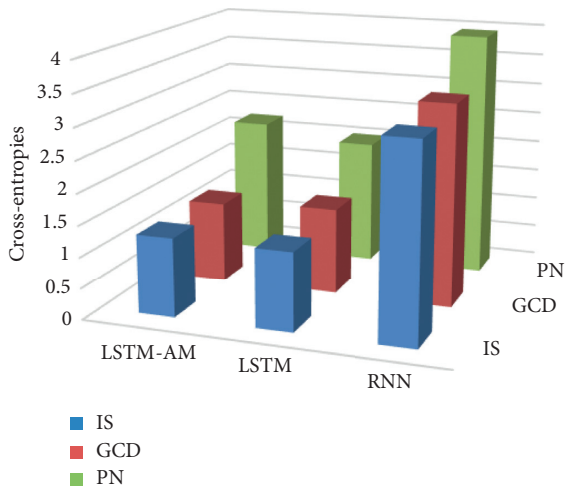


Figure 11: Cross-entropies of the 200-unit model using IS, GCD, and PN problems.

Table 6: Cross-entropy comparison of different models using all source codes.

Model	Units (neurons)			
	50	100	150	200
LSTM-AM	3.96	2.99	2.75	2.17
LSTM	3.89	3.26	2.50	2.31
RNN	5.11	4.36	3.87	3.53

generates error words and predicted words' probability. Since both the standard LSTM and the LSTM-AM networks identified source code errors quite well compared with the RNN and other networks when the 200-unit model was used, a 200-unit model was selected for use in all of our empirical experiments.

An erroneous source code sequence evaluated by the standard LSTM network is shown in Figure 12. Here, it can be seen that errors were detected in lines 2, 6, 15, and 16. In line 2, the word "*a*" in the "*gcd*" function was detected as an error candidate, after which the correct word was predicted to be ")" with a probability 0.62435395. The model decided that the "*gcd*" function might be without arguments, the word ")" was predicted instead of the word "*a*". In line 6, the error word is "if" and the predicted word is "else" with a probability 0.5808078. Additionally, in line 15, the predicted word is "blank space" in the place of "double quotation." Finally, in line 16, the model detected "*c*" as an error object and suggested with a high level of probability that it should be replaced by the word "*b*." The word "*c*" is irrelevant within the context of the program; it can be confirmed that the standard LSTM model successfully detected the error candidates shown in Figure 12, as listed in Table 7.

The same incorrect source code was then evaluated by the LSTM-AM network, as shown in Figure 13. The error locations are in lines 2, 15, and 16. The word "*a*" in the "*gcd*" function was detected as an error candidate and the predicted word ")" was suggested. In line 15, the word "double quotation" was identified as a bug, and the predicted word "blank space" was suggested. The word "*c*" in line 16 was recognized as an error and the corresponding predicted word suggested was "b" with a probability of 0.9863272, as shown in Table 8.

Another interesting erroneous source code, which exists in some logical errors, was evaluated by the standard LSTM network, as shown in Figure 14. All the detected error words and their corresponding predicted corrections of Figure 14 are listed in Table 9.

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

Figure 12: Erroneous source code with colored fonts and underlines evaluated by the standard LSTM.

Table 7: LSTM network error detection and predictions.

Erroneous words	Erroneous word's probability	Location	Predicted words	Probability
A	0.000496462	2	)	0. 6243539
If	0.014852316	6	else	0.5808078
“	0.029112648	15	space	0.6583209
c	$8.5894303e^{-10}$	16	*b*	0.9261215

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

Figure 13: Erroneous source code with colored fonts and underlines evaluated by the LSTM-AM.

Table 8: LSTM-AM network error detection and predictions.

Erroneous words	Erroneous word's probability	Location	Predicted words	Probability
A	0. 000309510	2	)	0. 5722154
“	0. 045484796	15	space	0. 7051629
c	$2.838025e^{-07}$	16	*b*	0.9863272

```
1 #include <stdio.h>
2
3 int      main(void)
4 {
5   int      x, y;
6   int      a, b;
7
8   int      rmd;
9   int      i;
10  int      rst;
11
12  scanf("%d %d",&a, &b);
13  if( a >= b )
14  {
15      x = a;
16      y = b;
17  }else
18  {
19      x = b;
20      y = a;
21  }
22
23  rmd = y % x;
24  if( rmd == 0 )
25  {
26      printf("%d\n",y);
27  }
28  else
29  {
30
31      for(i=1; i<=rmd; i++)
32      {
33          if((( y % i) == 0) && (( rmd % i) == 0))
34          {
35              rst = i;
36          }
37      }
38      printf("%d\n",rst);
39  }
40  return 0;
41 }
```

Figure 14: Logical erroneous source code with colored fonts and underlines assessed by the standard LSTM.

Table 9: Error words detection and prediction using standard LSTM.

Erroneous words	Erroneous word's probability	Location	Predicted words	Probability
for	0.049593348	31	*i*	0.1376049
rst	0.02372846	38	*rmd*	0.6147145
}	0.0470908	39	return	0.95013565

Similarly, the same erroneous source code was tested by the LSTM-AM network, as shown in Figure 15. The detailed error descriptions of Figure 15 are listed in Table 10, where it can be seen that the LSTM-AM network detected all of the potential errors, including the true logical errors, successfully.

6.3. Classification of Source Codes. We evaluated our proposed LSTM-AM and other benchmark models using both clean and erroneous source codes. For extensive experiments, we selected several benchmark models to compare classification results such as (i) Random Forest (RF) [36] method, (ii) Random Forest (RF) method trained with secret attributes by Restricted Boltzmann Machine (RBM) [37], and (iii) Random Forest (RF) method learned with secret attributes by Deep Belief Network (DBN) [38].

The precision, recall, and *F*-measure are expressed by equations (23)–(25), respectively, to verify the classification performances:

$$\text{precision}\,(P) = \frac{tp_{e\longrightarrow e}}{tp_{e\longrightarrow e} + fp_{c\longrightarrow e}}, \tag{23}$$

$$\text{recall}\,(R) = \frac{tp_{e\longrightarrow e}}{tp_{e\longrightarrow e} + fn_{e\longrightarrow c}}, \tag{24}$$

$$F\text{-measure} = \frac{2 * P * R}{P + R}, \tag{25}$$

where $tp_{e\longrightarrow e}$ is called true positive, the case $e \longrightarrow e$ means defective source code classifies as erroneous, and $fp_{c\longrightarrow e}$ is called false positive; the case $c \longrightarrow e$ means that the clean source code is classified as erroneous. The term $fn_{e\longrightarrow c}$ is called false negative where $e \longrightarrow c$ means that the erroneous source code is classified as a clean source code. *F*-measure is called the harmonic mean of recall and precision. Generally, we cannot achieve optimal results simultaneously for recall and precision. For example, if all the source codes are classified as defective, the resulting recall score will be 100% where the precision score will be small. Therefore, *F*-measure is a trade-off between recall and precision. The range of the *F*-measure score is between 1 and 0; the higher score implies a better classification model.

Under normal circumstances, our proposed language model detects all possible errors in source codes where all the detected errors are not true errors (TE). So, we considered only TE for the classification process. An error is called a TE when the predicted probability is more than 0.90. We aligned the term true positive $tp_{e\longrightarrow e}$ with our proposed model when the model detects TE in erroneous source codes. Again, in case of the term false positive $fp_{c\longrightarrow e}$, at least a single TE is detected within correct source codes.

```
1 #include <stdio.h>
2
3 int      main(void)
4 {
5   int      x, y;
6   int      a, b;
7
8   int      rmd;
9   int      i;
10   int      rst;
11
12    scanf("%d %d",&a, &b);
13    if( a >= b )
14    {
15        x = a;
16        y = b;
17    }else
18    {
19        x = b;
20        y = a;
21    }
22
23      rmd = y % x;
24    if( rmd == 0 )
25    {
25        printf("%d\n",y);
27    }
28    else
29    {
30
31          for(i=1; i<=rmd; i++)
32        {
33            if((( y % i) == 0) && (( rmd % i) == 0))
34            {
35                rst = i;
36            }
37        }
38        printf("%d\n",rst);
39    }
40    return 0;
41  }
```

Figure 15: Logical erroneous source code with colored fonts and underlines assessed by the standard LSTM-AM.

Finally, the term false negative $fn_{e\longrightarrow c}$, not a single TE, is detected in erroneous source code which means that it classifies the erroneous source code as clean code. As mentioned above, all the models are trained by using correct source codes and tested on 500 randomly chosen source codes from each problem set (IS, GCD, and PN). The classification results are listed in Tables 11–13 for the IS, GCD, and PN source codes, respectively.

In the classification process, the *F*-measure scores of the LSTM-AM and other state-of-the-art models are shown in Figure 16. *F*-measure results show that the classification performance of our proposed model is better than other methods.

7. Discussion

To assess our proposed intelligent support model, we defined three performance measurement indices such as error prediction accuracy (EPA), error detection accuracy (EDA), and model accuracy (MA), shown in equations (26) to (28). In particular, we evaluated our proposed model and other benchmark models using equation (28).

$$\text{EDA} = \frac{\text{True Errors (TE)}}{\text{Total Detected Errors (TDE)}} \times 100\%, \tag{26}$$

$$\text{EPA} = \frac{\text{True Correct Words (TCW)}}{\text{Total Predicted Words (TPW)}} \times 100\%, \tag{27}$$

$$\text{MA} = \frac{\text{EDA} + \text{EPA}}{2}. \tag{28}$$

In most cases, the proposed model detects potential errors in the codes. Among these errors, there are a few

Table 10: Error words' detection and prediction using the LSTM-AM network.

Erroneous words	Erroneous word's probability	Location	Predicted word	Probability
a	0.034574546	12	*x*	0.9269715
=	0.012642788	13	*b*	0.9468921
rmd	0.03553478	23	}	0.6362259
for	0.037460152	31	while	0.5292723
rst	0.025345348	35	*i*	0.8597483

Table 11: Classification performance comparison using insertion sort (IS) source codes.

Models	Precision (*P*)	Recall (*R*)	*F*-measure
LSTM-AM	0.99	0.97	0.97
LSTM	0.90	0.88	0.88
RNN	0.82	0.79	0.80
CNN	0.85	0.80	0.82
RF	0.62	0.55	0.58
RF + RBM	0.66	0.65	0.65
RF + DBN	0.71	0.66	0.68

Table 12: Classification performance comparison using the greatest common divisor (GCD) source codes.

Models	Precision (*P*)	Recall (*R*)	*F*-measure
LSTM-AM	0.98	0.95	0.96
LSTM	0.87	0.89	0.87
RNN	0.80	0.81	0.80
CNN	0.83	0.84	0.83
RF	0.64	0.59	0.61
RF + RBM	0.70	0.63	0.66
RF + DBN	0.75	0.80	0.77

Table 13: Classification performance comparison using prime number (PN) source codes.

Models	Precision (*P*)	Recall (*R*)	*F*-measure
LSTM-AM	0.95	0.94	0.94
LSTM	0.88	0.86	0.86
RNN	0.76	0.79	0.77
CNN	0.80	0.82	0.80
RF	0.59	0.60	0.59
RF + RBM	0.63	0.62	0.62
RF + DBN	0.65	0.66	0.65

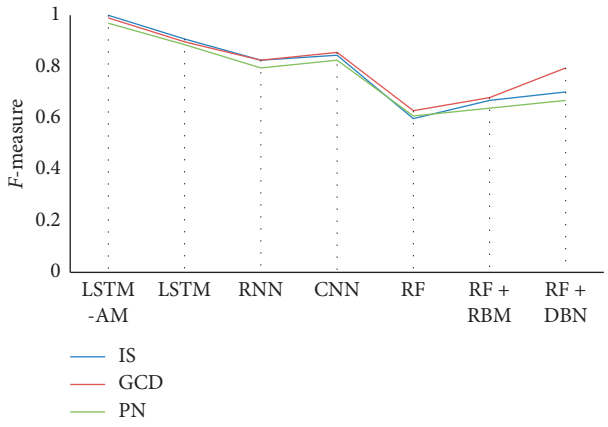


Figure 16: *F*-measure score comparison between the LSTM-AM and other state-of-the-art models.

original errors called true errors (TE). Similarly, out of the total predicted words, where some of the original correct words are left, they are called True Correct Words (TCW).

In the evaluation process, we discarded the RNN and other benchmark models because they obtained high cross-entropies whereas standard LSTM achieved very low cross-entropies. Therefore, we validated both the standard LSTM and LSTM-AM networks using several randomly chosen erroneous source codes. Figure 12 and Table 7 present the details of error detection and prediction by standard LSTM. The standard LSTM detected errors in lines 2, 6, 15, and 16 and provided the corresponding candidate words "*a*," "if," "double quotation," and *c*, respectively. The predicted correct words are "(," "else," "blank space," and "*b*." Although these results show that the standard LSTM had detected the most probable erroneous words and locations, not all of the candidate errors are true errors (TE). In line 2, the model

Table 14: The evaluation results of erroneous source code using standard LSTM.

Evaluation indices	Results (%)	Descriptions
EDA	25	TE = 1, TDE = 4
EPA	25	TCW = 1, TPW = 4
MA	25	EDA = 25, EPA = 25

Table 15: The evaluation results of erroneous source code using LSTM-AM.

Evaluation indices	Results (%)	Descriptions
EDA	33.33	TE = 1, TDE = 3
EPA	33.33	TCW = 1, TPW = 3
MA	33.33	EDA = 33.33, EPA = 33.33

Table 16: Evaluation results by the standard LSTM and LSTM-AM.

Model	EDA (%)	EPA (%)	MA (%)
LSTM	66	30	48
LSTM-AM	90	72	81

detects "*a*" as an error candidate by guessing that "*gcd*" is a function without arguments. Then, as a consequence, it predicts a close parenthesis ")" as the correct word. Similarly, in line 6, the model detected "if" as a candidate error word and predicted "else" as a corresponding correction. In this case, the model calculated that the word "if" started at line 3 and ended at line 5 and that the word after line 5 should be "else." As a result, the standard LSTM predicted the word "else" in line 6 instead of the word "if." However, both the error predictions in lines 2 and 6 were incorrect, even though they were hypothetically reasonable. It should be noted that the error candidate word "*c*" in line 16 is a true error (TE) and the predicted word "*b*" is correct. The evaluation results using the standard LSTM for the erroneous source code in Figure 12 are presented in Table 14.

In Figure 13, the LSTM-AM model detected a total of three errors in lines 2, 15, and 16, with the predicted corresponding correct words being ")," "blank space," and "*b*" respectively, as shown in Table 8. The evaluation results using the LSTM-AM for the erroneous source code in Figure 13 are presented in Table 15.

To further evaluate the performance of our proposed model, we then took a somewhat larger and more complex erroneous source code and verified it using both the standard LSTM and LSTM-AM networks, as shown in Figures 14 and 15, respectively. The erroneous source code contains a logical error in line 23. In this source code, two inputs were taken from the keyboard as "*a*" and "*b*" variables. The higher value was assigned to variable "*x*" and the lower value was assigned to variable "*y*." Initially, variable "*x*" was thought to be a dividend and variable "*y*" was designated as a divisor. However, line 23 was checked to find the initial greatest common divisor used by the modular arithmetic operator where the small valued variable "*y*" was considered to be a dividend, and the higher valued variable "*x*" was considered to be a divisor. By following the code sequence, the correct logic would be *x*%*y*. Based on that aspect, the LSTM-AM network identified the logical error correctly by considering the previous source code sequence, whereas the standard LSTM could not detect the logical error in line 23. The evaluation results for erroneous source codes in Figures 14 and 15 are listed in Table 16, where it can be seen that the LSTM-AM network performance was even better in the case of the long source codes and complex codes with logical or other errors.

In addition to the abovementioned source code evaluations and examples, we evaluated about 300 randomly chosen erroneous source codes using the LSTM and LSTM-AM models and found that their average accuracy values were approximately 31% and 62%, respectively. Those detailed statistics are shown in Table 17.

Unlike the examples used in this study, programs' lengths can vary widely, with many containing from 500 to 1000 lines of source code, or more. One thing all have in common is that when writing a program, numerous variables and functions may be declared in many lines previously. Therefore, an attention mechanism is needed to capture the long-term source code dependencies, as well as to evaluate source code errors correctly. Our experimental results have shown that the LSTM-AM model was much more successful for the longer sequenced source code than was the standard LSTM model, as shown in Figure 17.

Additionally, some syntax and logical errors in source codes cannot be identified by traditional compilers. In such cases, our proposed LSTM-AM-based language model can provide meaningful responses to learners and professionals that can be used for the source code debugging and refactoring process. This can be expected to save time when working to detect errors from thousands of lines of source code, as well as to limit the area that must be searched to find the errors. Furthermore, the use of this intelligent support model can assist learners and professionals to more easily find the logical and other critical errors in their source codes. Moreover, the classification accuracy of our proposed model

Table 17: Overview of the average evaluation statistical results using various erroneous source codes.

Name	No.	Models					
		LSTM			LSTM-AM		
		EDA	EPA	MA	EDA	EPA	MA
GCD	100	34.27	32.13	33.2	65.47	59.04	62.26
PN	100	28	31	29.5	64.6	57.3	60.95
IS	100	31.4	29.8	30.6	63.6	61	62.3
MA			**31.1**			**61.84**	

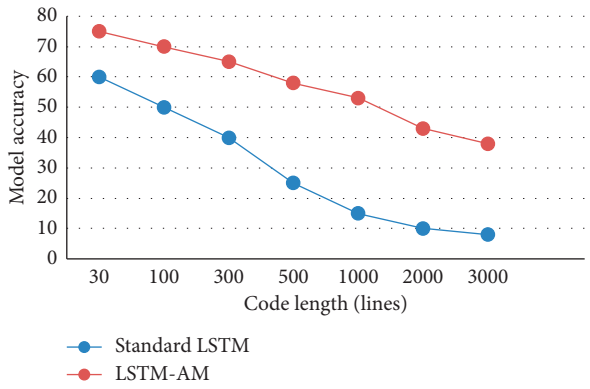


Figure 17: Accuracy of standard LSTM and LSTM-AM networks.

is much better than the other state-of-the-art models. The average precision, recall, and *F*-measure scores of the LSTM-AM model are 97%, 96%, and 96%, respectively, which outperformed other state-of-the-art models.

8. Conclusion

In the present research, we proposed an AI-based model to assist students and programmers in source code completion. Our proposed model is expected to be effective in providing end-to-end solutions for programming learners and professionals in the SE fields. The experimental results obtained in this study show that the accuracy of error detection and prediction using our proposed LSTM-AM model is approximately 62%, whereas standard LSTM model accuracy is approximately 31%. In addition, our approach provides the location numbers for the predicted errors, which effectively limits the area that must be searched to find errors, thereby, reducing the time required to fix large source code sequences. Furthermore, our model generates probable correction words for each error location and detects logical and other errors that cannot be recognized by conventional compilers. Also, the LSTM-AM model shows great success in source code classification than other state-of-the-art models. As a result, it is particularly suitable for application to long source code sequences and can be expected to contribute significantly to source code debugging and refactoring process. Despite the abovementioned advantages, our proposed model also has some limitations. For example, error detection and predictions are not always perfect, and the model sometimes cannot understand the semantic meaning of the source code because of the incorrect detection and predictions that have been produced. Thus, our future work will use a bidirectional LSTM neural network to improve this intelligent support model for source code completion.

Data Availability

We acquired all the training and test source codes from the AOJ system. Resources are accessed from the following websites through the API: https://onlinejudge.u-aizu.ac.jp/ and http://developers.u-aizu.ac.jp/index.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was funded by the JSPS KAKENHI (grant no. 19K12252).

References

[1] K. H. Dam, T. Tran, and T. Pham, "A deep language model for software code," 2016, https://arxiv.org/abs/1608.02715.

[2] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys*, vol. 51, no. 1, pp. 1–24, 2018.

[3] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: a case study (at google)," in *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, pp. 724–734, Hyderabad, India, May 2014.

[4] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "sk_p: a neural program corrector for MOOCs," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pp. 39-40, Amsterdam, Netherlands, November 2016.

[5] J. Pennington, R. Socher, and C. D. Manning, "Glove: global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532–1543, Doha, Qatar, October 2014.

[6] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pp. 269–280, New York, NY, USA, November 2014.

[7] M. White, C. Vendome, M. Linares-Vásquez, and P. Denys, "Toward deep learning software repositories," in *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR'15)*, pp. 334–345, Florence, Italy, May 2015.

[8] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu, "Advances in optimizing recurrent networks," in *Proceedings of the 2013 IEEE International Conference on Acoustics, Speech*

and Signal Processing, pp. 8624–8628, Vancouver, Canada, May 2013.

[9] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*, pp. 1556–1566, Beijing, China, July 2015.

[10] J. Reyes, D. Ramírez, and J. Paciello, "Automatic classification of source code archives by programming language: a deep learning approach," in *Proceedings of the 2016 International Conference on Computational Science and Computational Intelligence (CSCI)*, pp. 514–519, Las Vegas, NV, USA, December 2016.

[11] S. Bhatia, P. Kohli, and R. Singh, "Neuro-symbolic program corrector for introductory programming assignments," in *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, pp. 60–70, Gothenburg, Sweden, May 2018.

[12] M. Pedroni and B. Meyer, "Compiler error messages: what can help novices?," in *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, pp. 168–172, Portland, OR, USA, March 2008.

[13] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade, "DeepFix: fixing common c language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17)*, pp. 1345–1351, San Francisco, CA, USA, February 2017.

[14] M. M. Rahman, 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," *Applied Sciences*, vol. 10, no. 8, p. 2973, 2020.

[15] Y. Teshima and Y. Watanobe, "Bug detection based on lstm networks and solution codes," in *Proceedings of the 2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 3541–3546, Miyazaki, Japan, October 2018.

[16] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, pp. 1–15, San Diego, CA, USA, May 2015.

[17] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI'18).*, pp. 4159–4165, Stockholm, Sweden, July 2018.

[18] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 318–328, Prague, Czech Republic, July 2017.

[19] H. K. Dam, T. Pham, S. W. Ng et al., "Lessons learned from using a deep tree-based model for software defect prediction in practice," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 46–57, Montreal, Canada, May 2019.

[20] N.-Q. Pham, K. German, and G. Boleda, "Convolutional neural network language models," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 1153–1162, Austin, TX, USA, November 2016.

[21] G. Fan, X. Diao, H. Yu, Y. Kang, and L. Chen, "Software defect prediction via attention-based recurrent neural network," *Scientific Programming*, vol. 2019, Article ID 6230953, 14 pages, 2019.

[22] S. F. Chen and J. Goodman, "An empirical study of smoothing techniques for language modeling," in *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics (ACL'96)*, pp. 310–318, Santa Cruz, CA, USA, June 1996.

[23] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13)*, pp. 207–216, San Francisco, CA, USA, May 2013.

[24] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," 2012, https://arxiv.org/abs/1211.5063.

[25] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu, "Recurrent models of visual attention," in *Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS)*, pp. 2204–2212, Montreal, Canada, December 2014.

[26] T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1412–1421, Lisbon, Portugal, September 2015.

[27] J. Chen, H. Zhang, X. He, L. Nie, W. Liu, and T.-S. Chua, "Attentive collaborative filtering: multimedia recommendation with item- and component-level attention," in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'17)*, pp. 335–344, Shinjuku, Japan, August 2017.

[28] X. Ran, Z. Shan, Y. Fang, and C. Lin, "An LSTM-based method with attention mechanism for travel time prediction," *Sensors*, vol. 19, no. 4, p. 861, 2019.

[29] Y. Yoshizawa and Y. Watanobe, "Logic error detection system based on structure pattern and error degree," *Advances in Science, Technology and Engineering Systems Journal*, vol. 4, no. 5, pp. 1–15, 2019.

[30] T. Matsumoto and Y. Watanobe, "Towards hybrid intelligence for logic error detection," *Advancing Technology Industrialization Through Intelligent Software Methodologies, Tools and Techniques*, vol. 318, pp. 120–131, 2019.

[31] J. Cheng, L. Dong, and M. Lapata, "Long short-term memory-networks for machine reading," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 551–561, Austin, TX, USA, November 2016.

[32] Y. Watanobe, "Aizu Online Judge," 2018, https://onlinejudge.u-aizu.ac.jp/.

[33] Aizu Online Judge, "Developers site (API)," 2004, http://developers.u-aizu.ac.jp/index.

[34] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[35] D. P. Kingma and B. Jimmy, "Adam: a method for stochastic optimization," in *Proceedings of the 3rd International Conference for Learning Representations (ICLR)*, pp. 1–13, San Diego, CA, USA, May 2015.

[36] V. Svetnik, A. Liaw, C. Tong, J. C. Culberson, R. P. Sheridan, and B. P. Feuston, "Random forest: a classification and regression tool for compound classification and QSAR modeling," *Journal of Chemical Information and Computer Sciences*, vol. 43, no. 6, pp. 1947–1958, 2003.

[37] I. Sutskever, G. E. Hinton, and G. W. Taylor, "2*e* recurrent temporal restricted Boltzmann machine," in *Proceedings of the Advances in Neural Information Processing Systems*, pp. 1601–1608, Vancouver, Canada, December 2009.

[38] G. Hinton, "Deep belief networks," *Scholarpedia*, vol. 4, no. 5, p. 5947, 2009.