applied sciences

MDPI

Article

A Model with Iterative Trials for Correcting Logic Errors in Source Code

Taku Matsumoto *, Yutaka Watanobe * and Keita Nakamura

Graduate School of Computer Science and Engineering, The University of Aizu, Tsuruga, Ikki-machi, Aizu-Wakamatsu, Fukushima 965-8580, Japan; keita-n@u-aizu.ac.jp
* Correspondence: d8201105@u-aizu.ac.jp (T.M.); yutaka@u-aizu.ac.jp (Y.W.)

Abstract: It is difficult for students and teachers to detect and correct logic errors in source code. Compilers and integrated development environments (IDEs) have the ability to detect and correct syntax errors but it is also difficult for them to detect and correct logic errors. Although many machine learning approaches have been proposed that can show correction candidates for logic errors, they do not provide guidance concerning how the user should fix them. In this paper, we propose a model for correcting logic errors in a given source code. The proposed model realizes debugging of multiple logic errors in the source code by iterative trials of identifying the errors, correcting the errors, and testing the source code. In this model, in the first stage, a list of correction candidates is provided by a deep learning model, and then the list is given to an editing operation predictor that predicts the editing operation for the correction candidate. To learn the internal parameters of the proposed model, we use a set of solution codes created to solve the corresponding programming tasks in a real e-learning system. To verify the usefulness of the proposed model, we apply it to 32 programming tasks. Experimental results show that the correction accuracy is, on average, 58.64% higher than that of the conventional model without iterative trials.

Keywords: logic error correction; machine learning; deep learning; e-learning; online judge system; programming education

Citation: Matsumoto, T.; Watanobe, Y.; Nakamura, K. A Model with Iterative Trials for Correcting Logic Errors in Source Code. *Appl. Sci.* **2021**, *11*, 4755. https://doi.org/10.3390/app11114755

Academic Editor: Santi Caballé

Received: 30 March 2021
Accepted: 18 May 2021
Published: 22 May 2021

1. Introduction

Programming education has become indispensable for fostering engineers who will be active in today's advanced information society. One of the ways to develop programming skills is through coding exercises with repetitive problem-solving. Therefore, many e-learning systems have been developed to support programming education (see, for example, [1–3]). However, programming, which requires knowledge of syntax and logical thinking, is a difficult process, and learners still tend to stumble, especially in debugging. Among them, logic errors in compilable code are particularly troublesome. The logic error is a token in the source code that causes unexpected behaviors, which can be detected by runtime errors, wrong output results, and resource limit exceeded. So, since programming learners may have little knowledge and experience in programming, it is not easy for them to correct logic errors in source code. This is a problem faced not only by novice programmers but also by advanced programmers and instructors. If they cannot identify the logic errors in the source code, they will spend more time debugging and may lose their motivation in programming. Therefore, we believe that identifying logic errors in the source code can support programming learners.

To correct logic errors in source code, it is necessary to understand both the specifications of programming tasks and programming languages. The learner then needs to iterate over the debugging process until the program meets the specifications of the given programming task. Generally, in debugging work, logic errors are identified, corrected, and tested repeatedly. It is important for the testing to consider whether the expected results are obtained with acceptable time/space computational resources. Although a number of test

cases can enable the learner to confirm the existence of logic errors, it is often difficult to identify their location. Therefore, if the location of logic errors in the source code could be automatically identified and the errors then corrected, this would reduce the time required for debugging. Furthermore, additional support can be provided by identifying the true error from the correction candidates and showing the appropriate editing operations.

In this paper, to support programmers, especially for learners, we propose a debugging model for automatically correcting multiple logic errors in given source codes. This model realizes the detection of logic errors by learning the structure of solution codes stored in the database (DB) of a certain e-learning system. This system provides the user with many programming tasks, and the user can submit their solution codes. A set of test cases (and a special validator) is prepared for each programming task as judge data, and the automatic assessment system [4] rigorously tests whether the solution code meets the specifications of the corresponding task. Solution codes passed by the judge are considered as correct codes and other codes are incorrect with logic errors. The learning cycle using this system can be considered as test-driven development because it tests the source code using the test cases that are already prepared in the system. Our proposed model iteratively attempts to detect and correct errors and test the source code. The model consists of two submodels. The first is the Correct Code Model (CCM), which generates a sequence of tokens of correct code by learning the structure of a set of correct codes. By comparing the given source code with the prediction result, CCM can enumerate multiple correction candidates for logic errors. The second is the Editing Operation Predictor (EOP), which indicates the editing operation for the correction candidate obtained by CCM. To train and evaluate the proposed model, we employed solution codes accumulated in an online judge system [5] that provides many programming tasks and automatic assessments. The experimental results show that the correction accuracy of the proposed model is, on average, 58.64% higher across 32 tasks compared to the conventional model without iterative trials.

The contributions of this paper are as follows.

- Development of a debugging model that iteratively detects and corrects errors and tests the source code.
- Development of EOP with CCM that can predict the location of the errors, possible alterations as well as editing operations.
- Experiment of the proposed model conducted with real solution codes oriented to programming tasks obtained from an online judge system.
- Adaptation to use cases assuming that the user debugs incorrect code using the correction candidates and editing operations predicted by the proposed model. We also discuss the adaptation for both programming education and software engineering.

The rest of this paper is organized as follows. Section 2 introduces related research on logic errors and explores current issues. Section 3 describes the proposed model. Section 4 puts forward an experimental method for verifying the usefulness of the proposed model. Section 5 describes the experimental results and considerations. Section 6 concludes this paper.

2. Related Works

In this section, first, we should mention current approaches and methods for identifying and correcting errors focusing on educational scenes. We also consider related technologies in debugging oriented to software engineering and other purposes.

2.1. Approaches and Methods for Educational Scenes

In traditional learning methods in programming and software engineering exercises, learners can use compilers and IDEs to detect errors. However, as mentioned in the Introduction, to correct logic errors, learners need to take steps such as checking the execution results. Therefore, it is a burden not only for learners but also for instructors who have many students in the classroom.

Luxton-Reilley et al. report on existing approaches and future challenges in introductory programming [6]. While many debugging tools have been studied that focus on

syntax errors in source code, debugging tools that focus on logic errors have also been increasing. They also reported that although many debugging tools have been developed, the future issue is whether learners can understand the complexity of these debugging tools and debug them efficiently.

In order to help learners who cannot debug the source code using compilers and IDEs completely, many studies have been conducted to enhance the error messages output by them [7,8]. These studies reported that by analyzing the source code created by the learner and the error messages produced by the compiler, they were able to reduce the number of syntax errors encountered by the learner. However, since logic errors depend on the specifications of the programming task, they do not have the same rules as the syntax of a programming language. Therefore, debugging of logic errors considering the specifications of programming tasks is a major issue.

To support the debugging of logic errors by novice programmers in educational scenes, many studies have been conducted using source code groups created to meet the specifications of a certain task. Yoshizawa et al. proposed a static analysis method that considers the structure of the source code [9]. For this, the correct code group is converted into Abstract Syntax Trees (ASTs). The source code to be debugged is also converted to AST. By comparing the structure of the converted AST and the prepared ASTs, the position of the logic error and the type of the logic error can be obtained with high accuracy. In machine learning-based approaches, Teshima et al. proposed a CCM based on the LSTM Language Model (LSTM-LM) and correct codes [10]. LSTM-LM can indicate the position of logic errors in a given incorrect code and suggest possible words by learning the structure of the correct code. Rahman et al. improved this model by applying the Attention Mechanism with LSTM (LSTM-AttM) [11,12]. They also employed a bidirectional LSTM to detect logic errors and to present suggestions for corrections [13]. The models can also be applied to code completion [14]. Although the machine learning model used for error detection is different, logic error detection and correction are realized by learning the structure of the correct code.

These methods can predict the location of logic errors in the source code and their proposed fixes, but they cannot suggest to the learner how to correct the source code using this information. Moreover, even if the source code can be corrected using the predicted results, the corrected source code may still contain logic errors. Therefore, these methods cannot guarantee that the learner will be able to fix all the logic errors in the source code.

Our model can provide step-by-step information needed to debug logic errors: the location of the logic errors, its proposed fix, its editing operation, and whether the source code is correctable. This means that learners and instructors are given step-by-step opportunities to consider what kind of debugging they should carry out, depending on their proficiency in programming skills. Therefore, our model can provide pedagogical support that takes into account the learner's learning process, unlike the direct and immediate debugging support found in common software development tools.

2.2. Technologies in Debugging

Pre-trained models have been widely used to solve natural language processing tasks [15]. OpenAI proposed Generative Pre-Trained Transformer 3 (GPT-3) as state-of-the-art language model in 2020 [16]. The model is capable of generating sentences that are comparable to those written by humans. GPT-3 was realized by using a huge number of parameters of the model and a large amount of training data.

In recent years, machine learning techniques have been utilized to solve complex programming-related problems [17]. First, we should emphasize the recent evolution of IDE extensions. Functions such as error highlighting, completion, and refactoring are essential for software development. Such support can be realized by machine learning approaches with data analysis. For example, the latest technology in Visual Studio IntelliCode [18] has had a major impact on software engineering. In this approach, artificial intelligence uses

huge GitHub repositories for learning and provides intelligent completion capabilities that take into account not only listing variables and functions but also other situations.

Many automatic bug fixing methods have been proposed for quickly fixing software bugs [19]. Drain et al. realizes the detection and fix bugs using the standard transformer [20] as a language model based on source code extracted from GitHub repositories [21]. Ueda et al. have proposed the fix method by mining the editing histories in GitHub [22]. These methods make it possible to provide debugging support by using source codes, bug reports, edit histories, and so on.

Many methods have been proposed to fix source code using the correction candidates predicted by machine learning models. To correct multiple syntax errors in a source code, Gupta et al. proposed Deepfix [23], which iteratively corrects the errors in the source code. In this method, the source code can be modified line-by-line using one correction candidate predicted by the Sequence to Sequence (Seq2Seq) attention network. Hajipour et al. [24] proposed Samplefix, which iteratively corrects errors using a Seq2Seq model. Each time the source code is modified, these methods use a compiler to verify whether an error still exists. It was shown that the accuracy of correcting syntax errors in the source code can be improved by making repeated corrections. However, although source code verification using a compiler can correct syntax errors, it is difficult to find and correct logic errors.

Gupta et al. proposed a method for predicting logic error locations using the tree convolutional neural network and AST [25]. Experimental results showed that the percentage of source code that contains true errors in the lines predicted by this model was less than 30% when the number of candidate lines is one, and 80% when the number of candidate lines is 10. However, a large number of detection candidates can make it difficult for the user to find the true logic error. Therefore, there is a tradeoff between accuracy and increased number of detection candidates, as too high a number can cause the user to be overloaded with information. Vasic et al. proposed a program modification method leveraging a joint model using an Long Short-Term Memory (LSTM) network and an attention mechanism to solve the variable misuse problem [26]. Publicly available data were used to verify the accuracy of identifying and correcting variable misuse points. However, although variable misuse can be considered as one type of logic error, it cannot be used to identify other types of logic errors.

There is an approach called Fault Localization [27] that predicts the location of errors in the source code by using tests. The bug position is predicted using execution route information from the test cases. However, although it is possible to predict the location of the error using these methods, they do not provide instructions concerning how to correct the error. Lee et al. presented a method for correcting logic errors based on fault localization to support learners of functional languages [28].

Matsumoto et al. [29] considered a hybrid architecture of static analysis [9] and CCM for logic error detection [10]. They concluded that CCM is advantageous as it can detect more logic errors. Furthermore, they proposed a method for determining the threshold value for controlling the number of candidates indicated by the CCM to reduce the overload [30]. A set of incorrect codes was applied for CCM training to determine the threshold. The authors suggested that a threshold suitable for each individual programming task should be determined as this threshold can fluctuate for different tasks.

In summary, many methods have been proposed to detect errors in source code. Various researchers use machine learning-based models to detect errors in source code and use that information to modify the code. However, although there are many methods for correcting syntax errors, there are few debugging models for logic errors.

3. Proposed Model

This section introduces a proposed model using the LSTM-LM and Support Vector Machine (SVM). First, we explain CCM used to identify logic errors in the source code. Next, the proposed EOP will be described.

3.1. Overview of Proposed Model

Figure 1 outlines the proposed model for correcting multiple logic errors in a given source code. The model debugs by iteratively identifying and correcting logic errors and testing the modified code. This model consists of CCM and EOP. The EOP predicts the editing operation from the correction candidates obtained by CCM. Based on the obtained operations, EOP seeks to correct the logic error in the source code. Then, the model tests whether the modified source code is correct or not. If the source code is incorrect, it again becomes input data for CCM. In this way, the model can debug source code containing multiple logic errors.

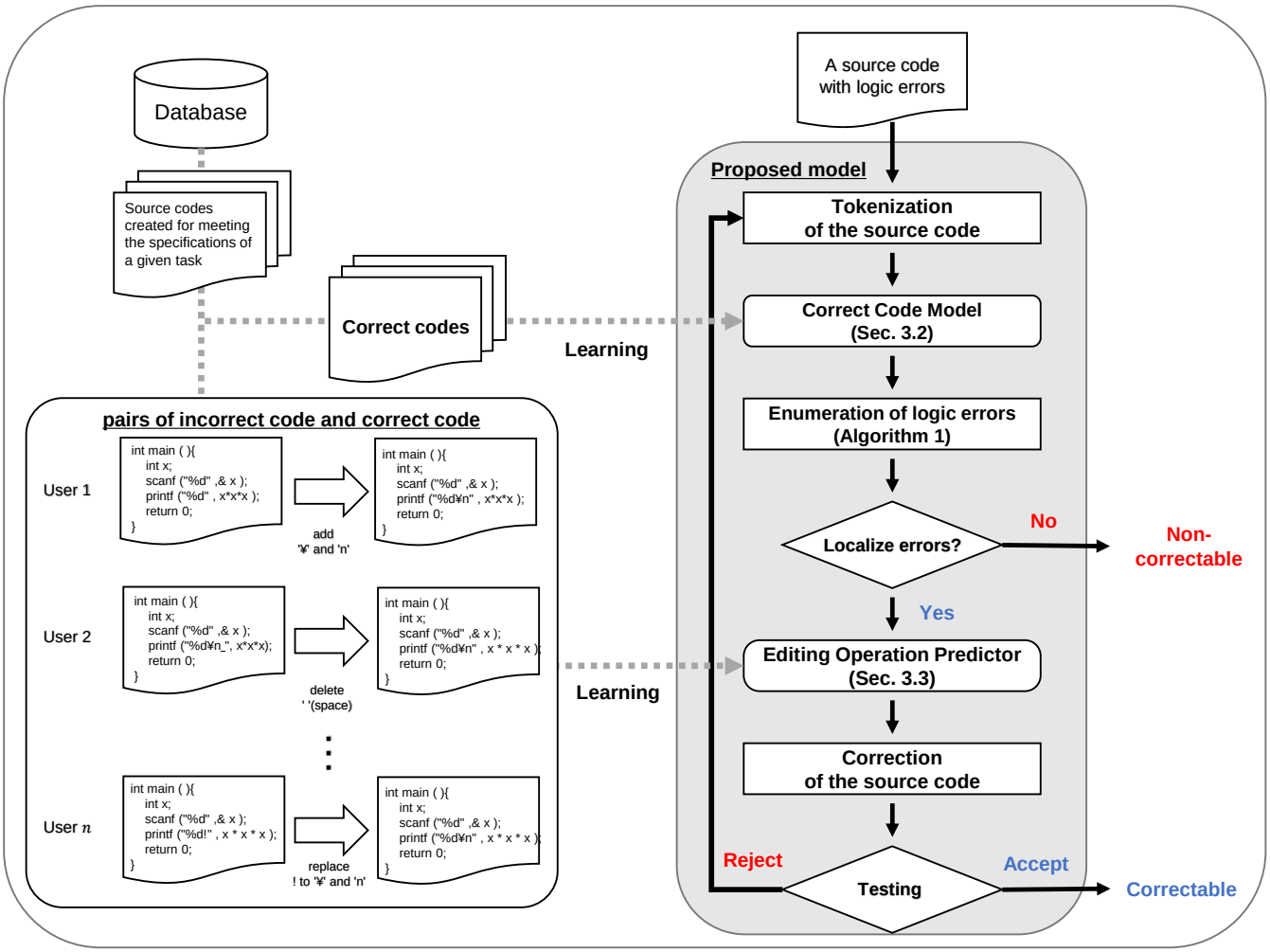

Figure 1. Overview of the proposed model.

To build the proposed model, a set of source codes oriented to the specification of a programming task is required. CCM is constructed by LSTM-LM, which has learned the structure of a set of correct codes to predict the position of logic errors. EOP can predict the editing operation for the correction candidate by learning the editing operation performed between the incorrect code and the corresponding correct code.

3.2. Correct Code Model (CCM)

3.2.1. Long Short-Term Memory Language Model (LSTM-LM)

The LSTM-LM is one of the RNNs that can handle time-series data. LSTMs [31] have the useful characteristic of being able to prevent gradient disappearance and gradient explosion. The language model [32] enables sentence generation and machine translation

based on the structure of the learned data. An LSTM-LM can be constructed by combining the following three layers: an embedding layer, an LSTM layer, and a softmax layer (Figure 2).

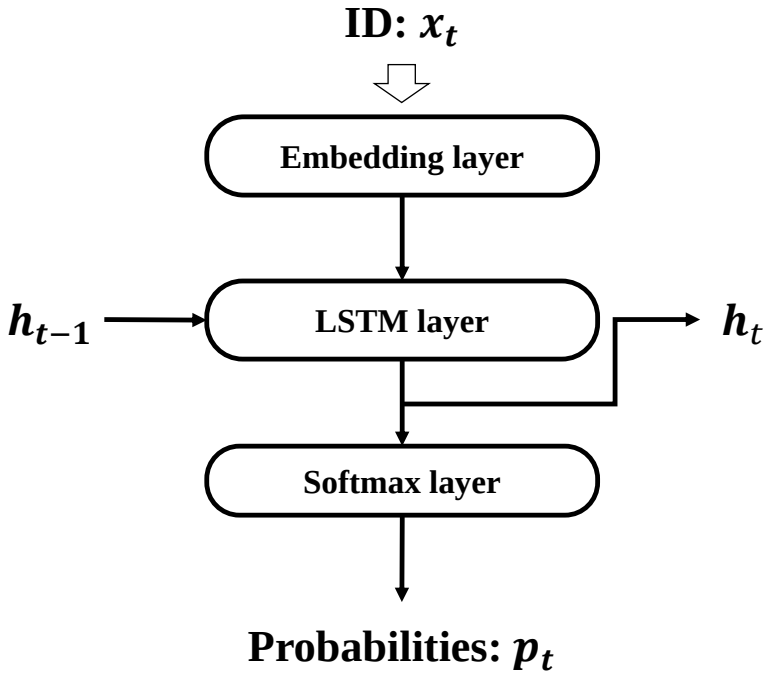

Figure 2. LSTM language model (LSTM-LM).

The source code is tokenized based on the vocabulary table. Variables, functions, reserved words, characters, and so on are considered vocabulary entries for each programming language. A unique identifier (ID) is assigned to each vocabulary entry. Since the token sequence is still a string, it is converted into a sequence of numerical values $\boldsymbol{x} = [x_1, x_2, \ldots, x_t, \ldots, x_n]$ based on the vocabulary table.

The embedding layer embeds a given input sequence $\boldsymbol{x}$ into the vector $\boldsymbol{e} = [e_1, e_2, \ldots, e_t, \ldots, e_n]$. Then, the LSTM layer uses the given vector $\boldsymbol{e}$ as an input to calculate the hidden output $\boldsymbol{h} = [h_1, h_2, \ldots, h_t, \ldots, h_n]$. The hidden output h_t obtained from the LSTM layer is converted into a word-by-word probability distribution $\boldsymbol{p}$ by the softmax layer.

CCM is constructed by learning the internal parameters of LSTM-LM using a set of correct codes. CCM learns the internal parameter by using the sequence $\boldsymbol{x}$ as the learning data. The probability $\boldsymbol{p_t}$ outputted by CCM indicates the probability distribution of the next token. By using the function argmax for the probabilities, CCM can output predicted sequence $\hat{\boldsymbol{x}} = [\hat{x_1}, \hat{x_2}, \ldots, \hat{x_t}, \ldots, \hat{x_n}]$ from the input sequence $\boldsymbol{x}$.

3.2.2. Localization of Logic Errors by CCM

Algorithm 1 localizes logic errors in a given source code by comparing the input sequence $\boldsymbol{x}$ with the prediction result obtained by CCM. If the sequence $\boldsymbol{x}$ follows the correct code, the token x_{t+1} at the time $t+1$ and the token $\hat{x}_{t+1}$ predicted by CCM are equal. On the other hand, if the sequence $\boldsymbol{x}$ is an incorrect code, the token x_{t+1} at the time $t+1$ and the token $\hat{x}_{t+1}$ predicted by CCM may be different. This means that the token x_{t+1} of the sequence is unlikely to appear as a sequence of tokens in the set of correct codes. Therefore, the token detected by Algorithm 1 and its position are likely to be logic errors. The procedure *ListCorrection* in the Algorithm 1 shows the positions of three tokens which are correction candidates: x_t, x_{t+1}, and $\hat{x}_{t+1}$.

Algorithm 1 *listCorrection* = Localization of logic errors ($\boldsymbol{x}$).

$\boldsymbol{p} \leftarrow CCM(\boldsymbol{x})$
listCorrection $\leftarrow$ []
for $t = 0 \ldots \boldsymbol{x}$. length **do**
  $\hat{x}_{t+1} \leftarrow \arg\max(\boldsymbol{p}_t)$
  if $x_{t+1} \neq \hat{x}_{t+1}$ **then**
    listCorrection.append([$\arg\min(\boldsymbol{p}_t), x_t, x_{t+1}, \hat{x}_{t+1}, t, t+1, t+2$])
listCorrection.sort() by probabilities
return *listCorrection*

3.3. Editing Operation Predictor (EOP)

If the token $\hat{x}_{t+1}$ predicted by Algorithm 1 and the original token x_{t+1} do not match, the source code needs to be edited using the predicted token $\hat{x}_{t+1}$ for the position of the token x_t and the token x_{t+1}. The editing operations are considered as follows using the three tokens.

insert insert the predicted token $\hat{x}_{t+1}$ between token x_t and the next token x_{t+1}.
delete delete the next token x_{t+1} between token x_t and $\hat{x}_{t+1}$.
replace replace the next token x_{t+1} with the predicted token $\hat{x}_{t+1}$.

Editing operations can be predicted by using the tokens x_t, x_{t+1} and $\hat{x}_{t+1}$. This assumes that the programmer can edit the source code if he/she knows the token and position to modify. EOP predicts the editing operation from the structure of the source code $\boldsymbol{x}$ and the three tokens x_t, x_{t+1} and $\hat{x}_{t+1}$.

Figure 3 shows how the model edits the source code based on the correction candidates obtained from Algorithm 1. The correction candidate with the highest possibility of a logic error is selected from the correction candidates list obtained from Algorithm 1. The selected correction candidate is converted into feature vectors using the vocabulary table used for tokenizing. Four feature vectors of the same size as the vocabulary table are created. They are a vector for the position of token x_t, a vector for the position of token x_{t+1}, a vector for the position of token $\hat{x}_{t+1}$, and a total number of each vocabulary in the source code. The concatenation of these four vectors is used as the EOP input.

Construction of EOP

Editing operations of source code can be classified into three categories: insertion, deletion, or replacement of tokens. Therefore, we considered the prediction of edit operations for source code as a multi-classification problem. To construct the EOP, we use SVM [33], which is often used to solve multi-classification problems.

To learn the internal parameters of EOP, feature vectors and teacher labels are extracted using a set of source codes obtained from the DB. The submission logs of all users are extracted from the set of source codes, and a set of pairs, each of which consists of a correct and an incorrect code, are employed as learning data. The extracted pair corresponds to an editing process from an incorrect code to a correct code. By calculating the Shortest Editing Script (SES) that provides the editing operations between two source codes, it is possible to obtain the editing operations for the candidates to be corrected. The SES is calculated from the incorrect and correct codes by dynamic programming. Incorrect code and correct code are converted into an ID sequence in advance using the vocabulary table. By comparing the ID sequence of the incorrect code with the position of the SES, it is possible to label what kind of editing operation the token in the SES can perform on the editing position. EOP learns these feature vectors and labels for editing operations using SVM.

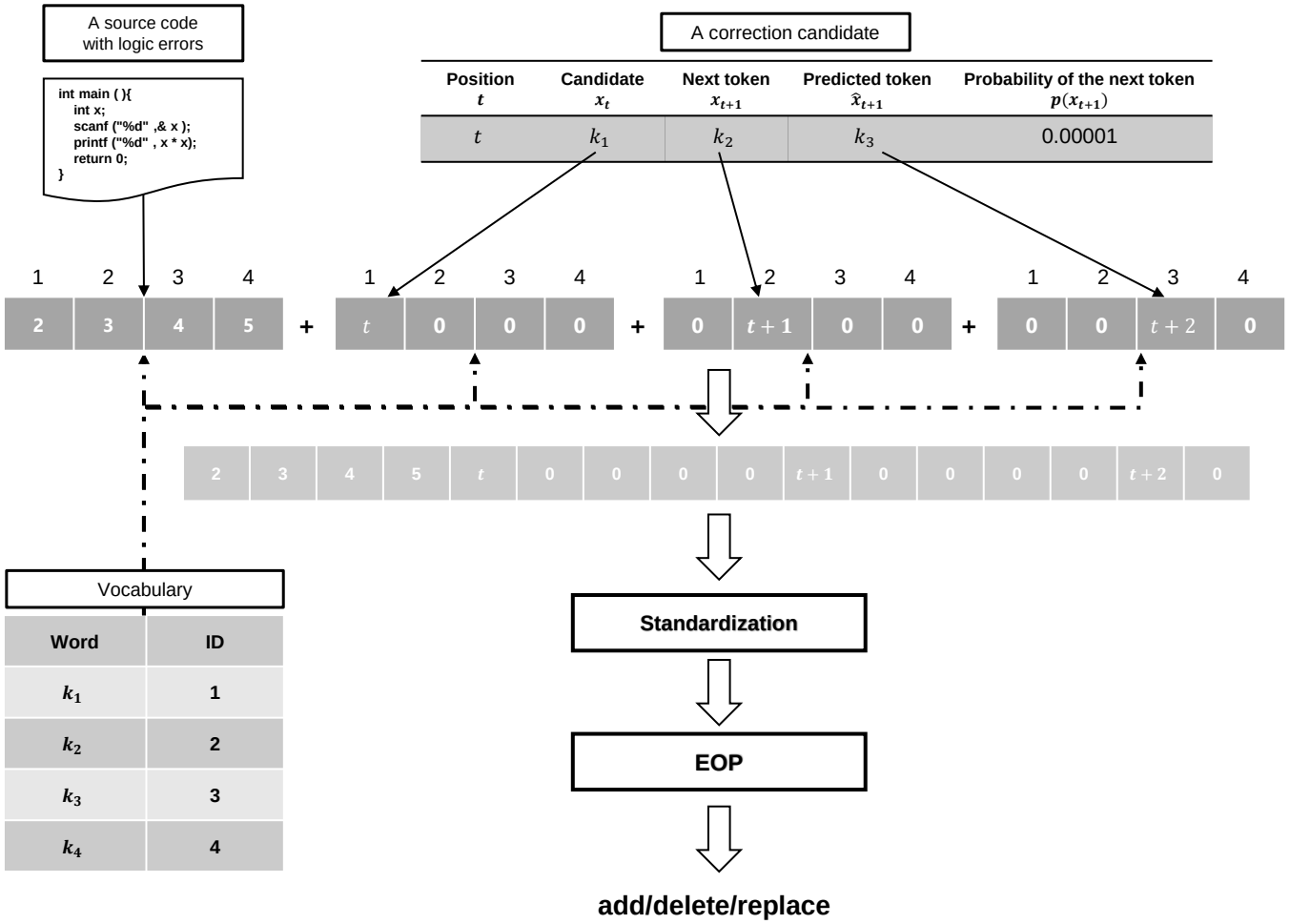

Figure 3. Role of the EOP.

3.4. Iterative Trials

The source code can be automatically corrected by combining CCM and EOP with Algorithm 1. If the source code modified by the model does not meet the specifications of the given programming task, it may need to be corrected again. In this model, the source code is tested using test cases to check whether the modified source code contains logic errors. If the modified source code passes the test, the debugging process is terminated assuming that all logic errors in the source code have been corrected. On the other hand, if the modified source code does not pass the test, the source code will be corrected again. If the target source code is written in a programming language that requires a compiler (e.g., C language), the model uses the compiler to convert it to an executable code before testing.

4. Experiment

To train the proposed model and conduct an experiment to verify its usefulness, we used source codes in C language that were written by programming learners for 32 programming tasks. We used a 64-bit Windows 10 computer with an Intel Core i9-9900K CPU (3.60 GHz), 32 GB RAM, and Nvidia GeForce RTX 2070 SUPER GPU.

The source codes used in the experiment were obtained from Aizu Online Judge (AOJ) [4,34,35], which is an e-learning system for programming. Table 1 shows details of the specifications of the 32 selected programming tasks. AOJ is a system that appeals to people of different abilities, from beginners to experts, and is currently used by more than 85,000 individuals. The system supports around 15 programming languages such as C/C++, Java, and Python. Users can freely select and challenge programming tasks from around 2500 tasks. They can submit their solution code to AOJ, and then it is automatically

judged by the AOJ system. This is so the user can know whether the source code they created meets the specifications of the selected programming task. About 5 million source codes submitted by users and their concomitant judgment results have been collected in AOJ's DB.

Table 1. Details of 32 selected tasks.

Task ID	Task Specification
ITP1_1_A	Outputs "Hello World" to standard output.
ITP1_1_B	Outputs the cube of a given integer x.
ITP1_1_C	Outputs the area and perimeter of a given rectangle.
ITP1_1_D	Receives an S seconds and converts it to $h : m : s$.
ITP1_2_A	Outputs small/large/equal relation of two given integers a and b.
ITP1_2_B	Receives three integers a, b, and c and outputs "Yes" if $a < b < c$, otherwise "No"
ITP1_2_C	Receives integers and outputs them in ascending order.
ITP1_2_D	Receives a rectangle and a circle, and determines whether the circle is arranged inside the rectangle.
ITP1_3_A	Outputs "Hello World" 1000 times.
ITP1_3_B	Receives an integer x and outputs it as is for multiple test cases.
ITP1_3_C	Receives two integers x and y, and outputs them in ascending order for multiple test cases.
ITP1_3_D	Receives three integers a, b, and c, and outputs the number of divisors of c in $[a, b]$
ITP1_4_A	Receives two integers a and b, and outputs a/b in different types.
ITP1_4_B	Receives a radius r, outputs the area and circumference of a circle.
ITP1_4_C	Receives two integers, a and b, and an operator op, and then outputs the value of $aopb$
ITP1_4_D	Receives a sequence of n integers a_i $(i = 1, 2, \ldots n)$, and outputs the minimum value, maximum value, and sum of the sequence.
ITP1_5_A	Draws a rectangle which has a height of H cm and a width of W cm. Draws the rectangle by '#'
ITP1_5_B	Draws a frame which has a height of H cm and a width of W cm.
ITP1_5_C	Draws a chessboard which has a height of H cm and a width of W cm.
ITP1_5_D	Structured programming without goto statement.
ITP1_6_A	Receives a sequence and output
ITP1_6_B	Given a deck of cards, finds any missing cards.
ITP1_6_C	Counts the number of elements in a three-dimensional array.
ITP1_6_D	Receives an $n \times m$ matrix $\boldsymbol{A}$ and an $m \times 1$ vector $\boldsymbol{b}$, and prints their product $\boldsymbol{Ab}$
ITP1_7_A	Receives a list of student test scores and evaluates the performance of each student.
ITP1_7_B	Identifies the number of combinations of three integers which satisfy 1) you should select three distinct integers from 1 to n, and 2) the total sum of the three integers is x.
ITP1_7_C	Receives the number of rows r, columns c, and a table of $r \times c$ elements, and prints a new table, which includes the total sum for each row and column.
ITP1_7_D	Receives an $n \times m$ matrix $\boldsymbol{a}$ and an $m \times l$ matrix $\boldsymbol{B}$, and prints their product, an $n \times l$ matrix $\boldsymbol{C}$.
ITP1_8_A	Converts uppercase/lowercase letters to lowercase/uppercase letters for a given string.
ITP1_8_B	Receives an integer and prints the sum of its digits.
ITP1_8_C	Counts and reports the number of each letter. Ignores characters.
ITP1_8_D	Finds a pattern p in a ring-shaped text s.

To focus on solution codes that include logic errors, we excluded those that could not be properly compiled because of syntax errors or warnings. Moreover, source codes including functions and function macros defined by users were excluded. Comments, tabs, and spaces in the source codes were also deleted to remove unnecessary tokens.

4.1. Training and Training Accuracy

The source code used for training was tokenized to the sequence $\boldsymbol{x}$ based on Table 2. In LSTM-LM, if the sequence $\boldsymbol{x}$ becomes long, memory leak may occur due to the increase in internal parameters. Therefore, we applied Hotelling's theory [36] to the length of the sequence $\boldsymbol{x}$ and statistically determined outliers. When the chi-square value of the length of each sequence exceeds 99% of the χ^2 distribution with one degree of freedom, the source code is regarded as an abnormal value. Source codes judged to be outliers were rejected

from the training data using the significance probability (p-value) $\chi^2_{0.99}(1) = 0.00016$. This means that source codes were rarely rejected.

Table 2. Vocabulary.

ID	Word	ID	Word	ID	Word	ID	Word	ID	Word	ID	Word
0	(masking value)	70	auto	92	’	114	=	126	T	160	m
1–20	(variables)	71	case	93	(	115	>	127	U	161	n
21–50	(functions)	72	char	94	)	116	@	128	V	162	o
51	continue	73	else	95	*	117	A	129	W	163	p
52	unsigned	74	enum	96	+	118	B	130	X	164	q
53	default	75	goto	97	,	119	C	131	Y	165	r
54	typedef	76	long	98	-	120	D	132	Z	166	s
55	define	77	main	99	.	121	E	133	[	167	t
56	double	78	void	90	/	122	F	134	\	168	u
57	extern	79	for	101	0	123	G	135	]	169	v
58	signed	80	int	102	1	124	H	136	^	170	w
59	sizeof	81	do	103	2	125	I	137	‘	171	x
60	static	82	if	104	3	126	J	138	a	172	y
61	struct	83	(space)	105	4	127	K	139	b	173	z
62	switch	84	!	106	5	128	L	140	c	174	{
63	return	85	?	107	6	129	M	141	d	175	|
64	break	86	_	108	7	130	N	144	g	176	}
65	const	87	"	109	8	131	O	155	h	177	∼
66	float	88	#	110	9	132	P	156	i		
67	short	89	$	111	:	133	Q	157	j		
68	union	90	%	112	;	134	R	158	k		
69	while	91	&	103	<	135	S	159	l		

CCM was constructed using Tensorflow (2.4.0) [37]. To train CCM, we set the batch size to 4, the number of hidden neurons to 256, and the number of epochs to 100 as hyperparameters. We selected categorical entropy as the loss function. To prevent overfitting of CCM, we used the Adam optimizer with four parameters based on the recommendation of [38]: learning rate $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 1e - 8$. As another countermeasure against overfitting, we set the dropout rate to be 0.5 based on the recommendation of [39].

Table 3 shows the accuracy of the CCM trained using the correct codes of 32 programming tasks. Task ID is the Problem ID of 32 programming tasks in AOJ, respectively. Number of training data is the number of correct codes used for training, excluding duplicate correct codes. Number of training data shows that solution codes accumulated in AOJ are available for each programming task.

Table 3. Training accuracy of CCM.

Task ID	# Training Data	Perplexity	Training Accuracy (%)
ITP1_1_A	73	1.08	93.28
ITP1_1_B	528	1.08	93.72
ITP1_1_C	1035	1.07	94.41
ITP1_1_D	1905	1.07	94.88
ITP1_2_A	886	1.06	96.24
ITP1_2_B	627	1.06	96.01
ITP1_2_C	1380	1.05	95.79
ITP1_2_D	1513	1.05	96.11
ITP1_3_A	380	1.04	94.10
ITP1_3_B	1241	1.07	94.09
ITP1_3_C	1530	1.06	95.05
ITP1_3_D	1266	1.06	95.29

Table 3. *Cont.*

Task ID	# Training Data	Perplexity	Training Accuracy (%)
ITP1_4_A	223	1.07	94.98
ITP1_4_B	212	1.06	94.01
ITP1_4_C	1148	1.05	96.61
ITP1_4_D	332	1.09	94.15
ITP1_5_A	946	1.06	96.17
ITP1_5_B	1673	1.07	95.82
ITP1_5_C	1712	1.05	95.44
ITP1_5_D	338	1.08	94.31
ITP1_6_A	1141	1.07	95.21
ITP1_6_B	760	1.07	95.19
ITP1_6_C	765	1.05	96.17
ITP1_6_D	1119	1.06	96.40
ITP1_7_A	934	1.05	96.50
ITP1_7_B	904	1.08	94.11
ITP1_7_C	769	1.09	94.98
ITP1_7_D	213	1.08	95.72
ITP1_8_A	569	1.08	94.02
ITP1_8_B	432	1.10	91.58
ITP1_8_C	455	1.04	94.21
ITP1_8_D	173	1.16	87.89

EOP was constructed using Scikit-learn (0.23.2) [40]. Ten-fold cross validation was used to evaluate the performance of EOP. Number of training data was divided into 10 parts, nine of which were used for training, and one was used as test data. From this process, 10 models were constructed and the accuracy of each of them was calculated.

Table 4 shows the accuracy of the trained EOP. Here, Task ID is the Problem ID of the 32 programming tasks. Number of training data is the number of features based on the editing information between the incorrect code and the corresponding correct code in the 32 programming tasks. We excluded duplicate data from number of training data. Training accuracy and test accuracy of EOP are averages of 10 models obtained using k-fold cross validation.

Since the training accuracy and verification accuracy is 80% or more in many programming tasks, the EOP is considered to have sufficient predictive performance for editing operations using the editing position and its token. This means that the EOP can likely be effectively used to correct incorrect code.

Table 4. Training and test accuracy of EOP.

Task ID	# Training Data	Training Accuracy (%)	Test Accuracy (%)
ITP1_1_A	1085	92.82	84.89
ITP1_1_B	5600	91.68	87.13
ITP1_1_C	5585	89.63	86.05
ITP1_1_D	5946	87.80	81.90
ITP1_2_A	6267	94.74	91.86
ITP1_2_B	3784	93.67	89.46
ITP1_2_C	16635	88.64	85.39
ITP1_2_D	7249	88.59	84.15
ITP1_3_A	1503	93.62	86.10
ITP1_3_B	6190	89.95	86.49
ITP1_3_C	8137	90.80	87.18
ITP1_3_D	2972	89.42	81.86

Table 4. *Cont.*

Task ID	# Training Data	Training Accuracy (%)	Test Accuracy (%)
ITP1_4_A	608	94.15	82.74
ITP1_4_B	891	96.53	87.88
ITP1_4_C	5286	90.36	85.60
ITP1_4_D	2331	85.59	79.58
ITP1_5_A	5829	94.87	91.13
ITP1_5_B	4936	90.83	85.96
ITP1_5_C	5400	90.65	85.89
ITP1_5_D	3560	93.53	98.99
ITP1_6_A	6393	94.73	91.69
ITP1_6_B	12291	86.17	81.26
ITP1_6_C	5586	90.59	85.97
ITP1_6_D	4654	86.16	80.28
ITP1_7_A	8937	89.75	85.48
ITP1_7_B	6718	88.97	83.22
ITP1_7_C	7805	87.85	82.93
ITP1_7_D	2125	90.01	84.28
ITP1_8_A	4316	89.58	82.95
ITP1_8_B	6700	82.52	75.30
ITP1_8_C	5080	86.56	79.92
ITP1_8_D	492	94.94	88.41

4.2. Evaluation

To verify the usefulness of the proposed model, we compared it with the conventional model without iterative trials. In the conventional model, only one trial of correction is performed using the correction candidates predicted by CCM. It is necessary to define metrics for evaluating the usefulness of the proposed model. We evaluated the performance of the proposed model by focusing on the correction accuracy of logic errors, the number of trials, and the execution time. We defined the correction accuracy as the ratio of the source codes in which all logic errors are corrected in the experimental data. We defined the number of corrections as the average number of corrections until the correct code is obtained in the experimental data. We defined the execution time as the time until the given source code becomes correct by the proposed model. However, if the model could not correct the given source code, it was not included in this result.

Table 5 shows the experimental data used to evaluate the proposed model. Here, targets is the number of incorrect codes that contain logic errors for each programming task. We selected the target codes with an edit distance of five or less and iterated until the code was corrected. Moreover, we categorized the source codes by the number of tokens that cause logic errors. The proposed model tries to correct logic errors until the given code is corrected. If the source code cannot be edited using the proposed model, modifications may be repeated infinitely as long as there are correction candidates. To avoid this situation, we set a termination condition in the proposed model. The model terminates its trials when the number of correction candidates indicated by Algorithm 1 becomes 0 or the number of iterations of the model exceeds 30.

Table 5. Details of experimental data.

Datasets		The Number of Logic Errors				
Task ID	**Targets**	**1**	**2**	**3**	**4**	**5**
ITP1_1_A	1524	281	1094	53	80	16
ITP1_1_B	630	42	536	13	31	8
ITP1_1_C	694	51	519	18	99	7
ITP1_1_D	500	16	365	20	88	11
ITP1_2_A	494	113	108	8	59	206
ITP1_2_B	727	17	466	15	160	69
ITP1_2_C	308	33	243	4	26	2
ITP1_2_D	350	28	72	11	236	3
ITP1_3_A	558	285	236	15	17	5
ITP1_3_B	336	105	175	18	27	11
ITP1_3_C	255	32	108	54	55	6
ITP1_3_D	379	123	202	15	35	4
ITP1_4_A	56	0	32	23	0	1
ITP1_4_B	42	8	7	7	7	13
ITP1_4_C	85	27	43	7	5	3
ITP1_4_D	63	10	30	6	17	0
ITP1_5_A	59	13	28	3	15	0
ITP1_5_B	132	10	93	2	26	1
ITP1_5_C	81	14	39	4	24	0
ITP1_5_D	50	8	24	3	14	1
ITP1_6_A	83	31	37	3	9	3
ITP1_6_B	70	41	20	5	1	3
ITP1_6_C	142	85	32	11	12	2
ITP1_6_D	89	24	44	4	17	0
ITP1_7_A	123	39	37	21	21	5
ITP1_7_B	183	8	41	5	127	2
ITP1_7_C	89	24	50	8	6	1
ITP1_7_D	23	2	6	1	14	0
ITP1_8_A	66	30	26	5	1	4
ITP1_8_B	46	15	10	6	14	1
ITP1_8_C	51	9	27	5	5	5
ITP1_8_D	5	2	1	0	2	0

5. Experimental Results and Discussion

5.1. Results

Table 6 shows the correction performance of the proposed model. The proposed model improves the correction accuracy of source code in all programming tasks. To ensure that the result was not due to a statistical chance, statistical tests were performed on the correction accuracy of the conventional and proposed models. The calculated p-value is 6.27×10^{-16}, which satisfies the general significance level of p-value < 0.05, indicating that these results are not chance results. The average correction accuracy of the conventional model for each programming task was 13.90%. On the other hand, the average correction accuracy of the proposed model was 72.55%. The average correction accuracy of the proposed model is 58.64% higher than the conventional model without iterative trials. This shows that the accuracy of correcting logic errors can be improved substantially by using the proposed model that introduces iterative trials. This means that the proposed model was able to correct hidden logic errors that could not be detected in the first trial.

Table 6. Correction performance.

Datasets		Correction Accuracy (%)		Number of Edits		Execution Time (s)		
Task ID	**Targets**	**Conventional Model**	**Proposed Model**	**Users**	**Proposed Model**	**Average**	**Min**	**Max**
ITP1_1_A	1524	18.11	99.54	2.00	2.12	0.38	0.23	2.62
ITP1_1_B	630	5.08	97.14	2.15	6.72	0.82	0.25	4.51
ITP1_1_C	694	4.18	93.08	2.44	4.87	0.60	0.22	4.27
ITP1_1_D	500	4.00	88.60	2.74	3.01	0.51	0.34	5.17
ITP1_2_A	494	24.09	89.07	3.68	9.85	1.47	0.25	4.28
ITP1_2_B	727	1.38	93.95	2.90	4.33	0.56	0.26	5.12
ITP1_2_C	308	6.49	72.73	2.88	4.82	0.52	0.27	3.82
ITP1_2_D	350	9.43	79.14	4.20	4.08	0.86	0.54	3.67
ITP1_3_A	558	30.47	97.49	1.65	4.98	0.62	0.20	8.10
ITP1_3_B	336	23.81	85.42	2.34	2.91	0.38	0.21	3.44
ITP1_3_C	255	6.27	69.41	3.73	5.08	0.58	0.21	8.11
ITP1_3_D	379	7.65	94.46	2.04	2.45	0.60	0.46	3.75
ITP1_4_D	63	7.94	77.78	3.18	3.39	0.74	0.48	5.16
ITP1_4_B	42	16.67	64.29	5.04	5.00	0.55	0.23	5.34
ITP1_4_C	85	14.12	80.00	2.49	3.66	0.70	0.27	6.38
ITP1_4_D	63	7.94	77.78	3.18	3.39	0.74	0.48	5.16
ITP1_5_A	59	27.12	83.05	2.82	3.06	0.42	0.26	1.58
ITP1_5_B	132	3.03	66.67	3.53	5.62	0.45	0.23	3.50
ITP1_5_C	81	3.70	60.49	4.08	6.16	0.47	0.26	3.44
ITP1_5_D	50	16.00	80.00	3.15	3.30	0.63	0.31	4.71
ITP1_6_A	83	16.87	81.93	2.43	5.04	0.56	0.26	1.90
ITP1_6_B	70	27.14	44.29	3.71	4.32	0.35	0.27	3.24
ITP1_6_C	142	16.90	58.45	2.89	8.11	0.73	0.27	6.09
ITP1_6_D	89	19.10	56.18	3.84	4.58	0.49	0.46	4.01
ITP1_7_A	123	22.76	69.92	3.31	3.45	0.56	0.26	10.07
ITP1_7_B	183	1.09	64.48	5.28	6.36	0.63	0.27	7.26
ITP1_7_C	89	6.74	26.97	7.38	3.79	0.17	0.28	2.10
ITP1_7_D	23	13.04	47.83	6.64	2.91	0.41	0.48	1.66
ITP1_8_A	66	18.18	40.91	4.48	2.56	0.16	0.26	1.90
ITP1_8_B	46	6.52	34.78	7.12	3.81	0.44	0.26	2.73
ITP1_8_C	51	47.06	72.55	3.32	3.00	0.38	0.26	2.04
ITP1_8_D	5	20.00	60.00	4.00	3.00	0.56	0.68	1.06
Average		13.90	72.55	3.57	4.42	0.56	0.30	4.16

We compared the number of edit operations in the proposed model with the number of edit operations by the user. The number of edit operations by the user is the edit distance between the experimental data created by each user and the corresponding correct code. The average number of edit operations of the proposed model is slightly larger than that of the user. This means that the correction candidates indicated by the proposed model include correction candidates that do not need to be edited.

The experimental results show the average, minimum, and maximum execution time of the proposed model for each programming task. The average execution time for all programming tasks is less than 1.5 (s). At the shortest, a given code can be corrected within 0.2 (s). At the longest, a given code can be corrected in 10.07 (s). These results show that the proposed model can debug logic errors in the source code within a reasonable timeframe.

Table 7 shows the correction accuracy for each number of logic errors in the source code for all programming tasks. The proposed model can correct all logic errors in the source code, even if there are multiple logic errors. This means that any logic errors that could not be detected in a first attempt were corrected in a later attempt. Therefore, this shows that the proposed model, which leverages iterative trials, is suitable as a debugging model for correcting multiple logic errors.

Table 7. Correction performance for each number of logic errors of all problems.

Datasets		Correction Performance			
The Number of Logic Errors	**Targets**	**Conventional Model (%)**		**Proposed Model (%)**	
1	1526	898	(58.8)	1304	(85.5)
2	4751	130	(2.7)	4305	(90.6)
3	373	6	(1.6)	291	(78.0)
4	1250	7	(0.6)	962	(77.0)
5	393	1	(0.3)	321	(81.7)

5.2. Limitations

These results show that the correction performance of the proposed model is high. However, focusing on detection performance and the number of trials, there is still room for improvement. We defined detection performance as the percentage of source codes in which a true logic error exists among the correction candidates obtained in the first trial. In the proposed model, a correction candidate most likely to be a logic error is selected and corrected. This means that if the top k correction candidates include true logic errors, it will be easier to correct those errors. Table 8 shows the detection performance of the proposed model. Task ID and Targets are the ID of each programming task and the number of experimental data. This shows the detection performance when the number of correction candidates is narrowed down to the top k. Where top $k = \infty$, this corresponds to all the correction candidates enumerated by the proposed model.

The detection performance of the proposed model is 95.54% when the top $k = \infty$. On the other hand, when the correction candidates are narrowed down to the top one, the true logic error can sometimes be missed. This means that true logic errors are less likely to appear when the correction candidates are narrowed down to the top k. The probabilities obtained from CCM are sufficient as a metric for detecting true logic errors in the source code. However, the probabilities may not be sufficient as a metric for selecting a correction candidate. The correction candidates indicated by CCM are likely to be logic errors, but they are not always logic errors. To navigate this issue, one approach could be to narrow down the correction candidates by analyzing what kind of logic errors are likely to occur in each programming task.

Table 8. Detection performance.

Datasets		Top-k (%)			
Task ID	**Targets**	$k = \infty$	$k = 1$	$k = 2$	$k = 3$
ITP1_1_A	1524	99.34	50.72	46.85	38.32
ITP1_1_B	630	99.84	90.79	85.08	43.17
ITP1_1_C	694	98.27	55.04	41.35	26.22
ITP1_1_D	500	99.60	47.00	28.40	8.20
ITP1_2_A	494	100.00	34.62	23.08	9.92
ITP1_2_B	727	100.00	54.61	44.70	30.12
ITP1_2_C	308	98.38	35.39	25.00	12.99
ITP1_2_D	350	98.57	24.00	9.43	2.57
ITP1_3_A	558	99.82	74.91	65.41	40.32
ITP1_3_B	336	98.51	45.54	22.92	7.44
ITP1_3_C	255	96.08	40.39	21.96	8.24
ITP1_3_D	379	100.00	57.26	33.77	14.78
ITP1_4_A	56	100.00	57.14	41.07	21.43
ITP1_4_B	42	92.86	19.05	11.90	2.38
ITP1_4_C	85	97.65	40.00	20.00	5.88
ITP1_4_D	63	98.41	12.70	3.17	0.00
ITP1_5_A	59	96.61	37.29	23.73	8.47
ITP1_5_B	132	96.21	19.70	9.09	1.52
ITP1_5_C	81	93.83	25.93	16.05	4.94
ITP1_5_D	50	100.00	14.00	6.00	2.00
ITP1_6_A	83	100.00	22.89	9.64	4.82
ITP1_6_B	70	90.00	7.14	2.86	1.43
ITP1_6_C	142	100.00	28.17	10.56	3.52
ITP1_6_D	89	96.63	10.11	3.37	1.12
ITP1_7_A	123	98.37	17.89	8.13	3.25
ITP1_7_B	183	98.91	17.49	14.21	7.10
ITP1_7_C	89	95.51	26.97	21.35	11.24
ITP1_7_D	23	100.00	4.35	0.00	0.00
ITP1_8_A	66	87.88	22.73	12.12	3.03
ITP1_8_B	46	95.65	26.09	21.74	13.04
ITP1_8_C	51	100.00	5.88	0.00	0.00
ITP1_8_D	5	100.00	0.00	0.00	0.00
Average		95.54	33.85	22.70	11.12

5.3. Use-Cases

Figure 4 shows an example of logic error correction using the proposed model for a sample programming task. The selected programming task is to find and output the cube of a given integer. An example of incorrect code shown in the upper left of the figure outputs the square of the given integer. The logic error in this source code is "x*x" on the fifth line. Therefore, if "*x" is inserted in the source code, the source code will become correct.

The table on the upper right in Figure 4 lists the prediction results for one application of the proposed model to the incorrect code. The candidate with the lowest probability of occurrence for the next word is the position of ")", and the proposed revision is "*". This demonstrates that the model can correctly predict this logic error. In addition, the proposed model predicts "insert * between x and)" from this information. The table at the bottom right of the figure shows the correction candidates when the proposed model is applied a second time to the source code that has already been corrected once. The candidate with the lowest probability of appearance is the position of ")", and the proposed amendment is "x". From the obtained information, the overall prediction by the proposed model is thus "insert x between * and)". Therefore, it is possible to correct all logic errors in this source code by applying the proposed model.

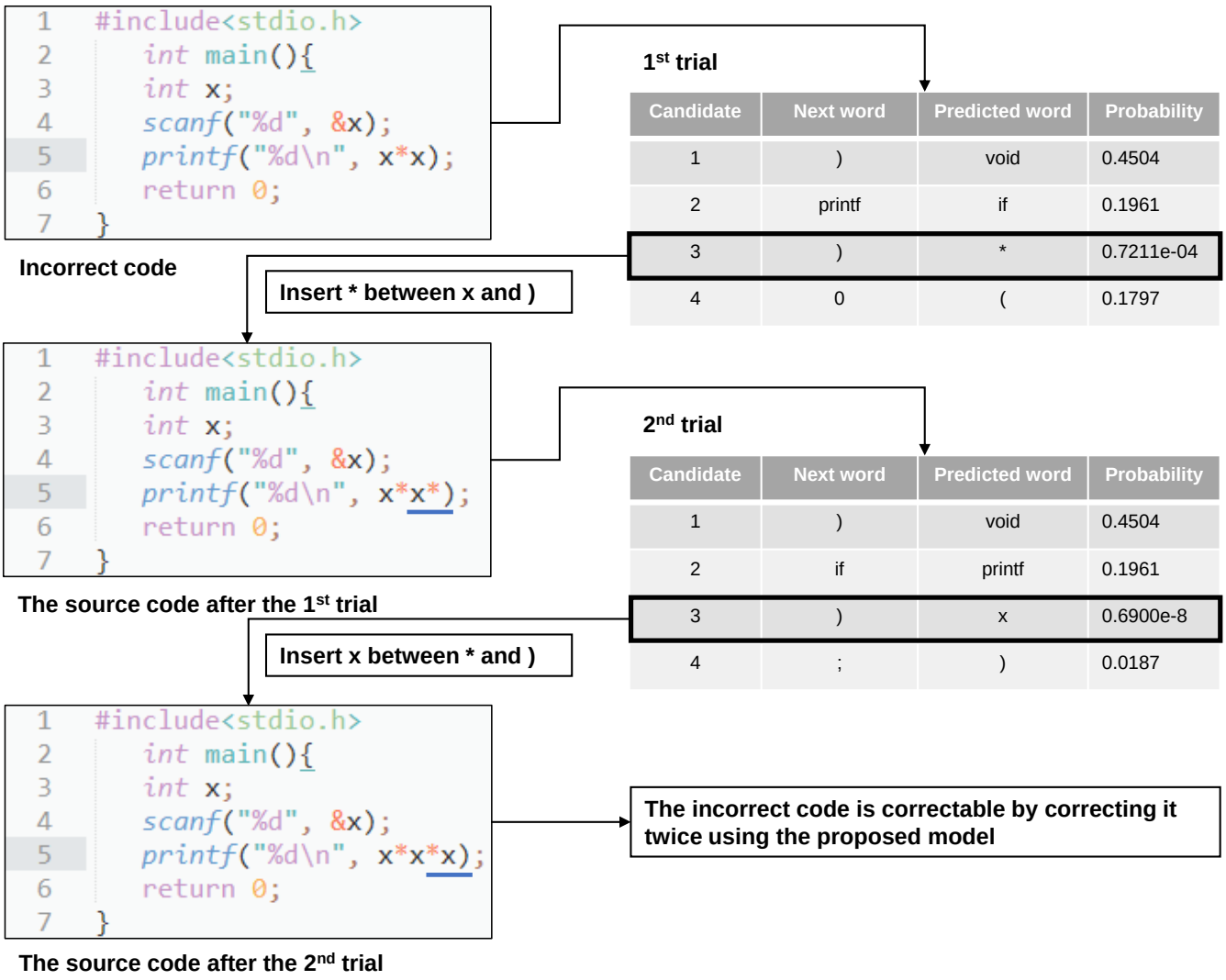

Figure 4. Example of logic error correction using proposed model.

In contrast, for the conventional model without iterative trials, the source code is modified using only the correction candidates obtained from the first trial. In the first trial, "insert *" between "x" and ")" is performed, as in the case of the proposed model. However, since "x" is not present in the correction candidates in this first trial, not all logic errors in the source code can be corrected. Therefore, the conventional model can only correct some and not all of the logic errors in the incorrect code.

The aim of debugging support in programming education is to provide learners with hints for correcting their source codes. However, giving too many hints to the learner can be problematic. Yi et al. [41] reported that novice programmers do not know how to modify programs efficiently using hints for correcting errors in the source code. This means that the hints must be adjusted according to the skills of the learner. Therefore, we suggest that it is possible to help individuals learn how to debug by showing not only correction candidates but also their editing operations.

In the proposed model, the source code is automatically modified based on the results obtained from each machine learning model. The modification of the source code can be replaced as a process to be performed by the user. Gradual hints related to the correction candidates and their editing operations by the proposed model can give the learners opportunities to think. Therefore, the quality of hints can be controlled by displaying information obtained from our model according to the programming proficiency of the learner.

Novice programmers need debugging support in many situations. There are situations where such individuals do not know what or how to debug when they get the verdict that the source code is incorrect. The proposed model can show novice programmers whether the source code can be corrected by iterative modification. When the source code can be corrected, the correction candidates, the editing operations, and the number of edits obtained in each trial can be presented as hints to these individuals. They can then use these hints to correct the source code. On the other hand, debugging support for intermediate programmers can be achieved by giving partial hints obtained from the proposed model. For example, it may be sufficient to disclose only the position of the logic error as a hint. In this way, the intermediate learner needs to think about the editing operation for the given position. Instructors and other expert programmers could, for example, be provided with feedback from the proposed model with all details at the very beginning, and then use this to help students at their discretion depending on their assessment of individual needs. To apply the proposed model and obtained feedback to educational sites, we should carefully consider learning efficiency. Generally, the feedback should not be direct and immediate supports such as those of conventional IDEs so that we can provide learners with chances to think and try to resolve the problem by themselves. The degree of such support should be controlled by learners or instructors according to their experience and learning modes.

The proposed model can also be used for software development. Generally, software consists of modules, packages, and subroutines. Implementations of these subroutines carry out numerical calculations and algorithm-level implementations to meet the specifications of a given task. It is necessary to use testing to verify whether the implemented subroutine is correct for the corresponding specification. If there is a programming task with the same or similar specifications as the subroutine implemented in a certain educational system, our model can be employed to modify the source code.

6. Conclusions

In this paper, we proposed a debugging model for correcting logic errors in a given source code. The model could correct multiple logic errors by repeatedly identifying and correcting errors, and testing the source code. In the experiment, to verify the advantage of the proposed model, 32 programming tasks and the corresponding solution codes in an online judge system were applied. By comparing the proposed model with another model without iterative trials, the results showed that the correction accuracy of the proposed model improved by 58.64% on average. In addition, this model can suggest operations for fixing code depending on the features around the detection location in the process of the iterative trials. The proposed model can also control the granularity of hints according to the proficiency of programmers and learners. Therefore, the proposed model takes into account educational effectiveness and can be applied to e-learning systems that support education not only in programming, but also in related subjects.

In future work, to improve the correction performance of our model, we would like to analyze what logic errors are likely to occur in each programming task. By comparing the results of this analysis with the candidates, we expect to increase the likelihood that the correcting candidates contain true logic errors.

Author Contributions: Conceptualization, T.M., Y.W. and K.N.; methodology, T.M. and Y.W.; software, T.M.; validation, T.M.; formal analysis, T.M.; investigation, T.M. and Y.W.; resources, T.M.; data curation, T.M.; writing—original draft preparation, T.M.; writing—review and editing, T.M., Y.W. and K.N.; visualization, T.M.; supervision, Y.W. and K.N.; funding acquisition, Y.W. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Japan Society for the Promotion of Science (JSPS) under KAKENHI grant number 19K12252.

Data Availability Statement: We collected all the training and test datasets from the Aizu Online Judge (AOJ) system. The resources were accessed through the APIs for the websites https://onlinejudge.u-aizu.ac.jp/, accessed on 21 May 2021 and http://developers.u-aizu.ac.jp/index, accessed on 21 May 2021.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Staubitz, T.; Klement, H.; Renz, J.; Teusner, R.; Meinel, C. Towards practical programming exercises and automated assessment in Massive Open Online Courses. In Proceedings of the 2015 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE), Zhuhai, China, 10–12 December 2015; pp. 23–30.
2. Crow, T.; Luxton-Reilly, A.; Wuensche, B. Intelligent tutoring systems for programming education: A systematic review. In Proceedings of the 20th Australasian Computing Education Conference, Brisbane, Australia, 30 January–2 February 2018; pp. 53–62.
3. Wasik, S.; Antczak, M.; Badura, J.; Laskowski, A.; Sternal, T. A survey on online judge systems and their applications. *ACM Comput. Surv. (CSUR)* **2018**, *51*, 1–34. [CrossRef]
4. Watanobe, Y.; Intisar, C.; Cortez, R.; Vazhenin, A. Next-Generation Programming Learning Platform: Architecture and Challenges. In *SHS Web of Conferences*;lEDP Sciences: Les Ulis, France, 2020; Volume 77, p. 01004.
5. Watanobe, Y. Aizu Online Judge. Available online: https://onlinejudge.u-aizu.ac.jp/ (accessed on 30 March 2021).
6. Luxton-Reilly, A.; Albluwi, I.; Becker, B.A.; Giannakos, M.; Kumar, A.N.; Ott, L.; Paterson, J.; Scott, M.J.; Sheard, J.; Szabo, C. Introductory programming: A systematic literature review. In Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education, Larnaca, Cyprus, 2–4 July 2018; pp. 55–106.
7. Becker, B.A.; Goslin, K.; Glanville, G. The effects of enhanced compiler error messages on a syntax error debugging test. In Proceedings of the 49th ACM Technical Symposium on Computer Science Education, Baltimore, MD, USA, 21–24 February 2018; pp. 640–645.
8. Denny, P.; Luxton-Reilly, A.; Carpenter, D. Enhancing syntax error messages appears ineffectual. In Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education, Uppsala, Sweden, 22–24 June 2014; pp. 273–278.
9. Yoshizawa, Y.; Watanobe, Y. Logic Error Detection System based on Structure Pattern and Error Degree. *Adv. Sci. Technol. Eng. Syst. J.* **2019**, *4*, 1–15. [CrossRef]
10. Teshima, Y.; Watanobe, Y. Bug detection based on LSTM networks and solution codes. In Proceedings of the 2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC), Miyazaki, Japan, 7–10 October 2018; pp. 3541–3546.
11. Rahman, M.M.; Watanobe, Y.; Nakamura, K. Source Code Assessment and Classification Based on Estimated Error Probability Using Attentive LSTM Language Model and Its Application in Programming Education. *Appl. Sci.* **2020**, *10*, 2973. [CrossRef]
12. Rahman, M.M.; Watanobe, Y.; Nakamura, K. A Neural Network Based Intelligent Support Model for Program Code Completion. *Sci. Program.* **2020**, *2020*. [CrossRef]
13. Rahman, M.M.; Watanobe, Y.; Nakamura, K. A Bidirectional LSTM Language Model for Code Evaluation and Repair. *Symmetry* **2021**, *13*, 247. [CrossRef]
14. Terada, K.; Watanobe, Y. Code Completion for Programming Education based on Recurrent Neural Network. In Proceedings of the 2019 IEEE 11th International Workshop on Computational Intelligence and Applications (IWCIA), Hiroshima, Japan, 9–10 November 2019; pp. 109–114.
15. Qiu, X.; Sun, T.; Xu, Y.; Shao, Y.; Dai, N.; Huang, X. Pre-trained models for natural language processing: A survey. *Sci. China Technol. Sci.* **2020**, *63*, 1872–1897. [CrossRef]
16. Brown, T.B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language models are few-shot learners. *arXiv* **2020**, arXiv:2005.14165.
17. Allamanis, M.; Barr, E.T.; Devanbu, P.; Sutton, C. A survey of machine learning for big code and naturalness. *ACM Comput. Surv. (CSUR)* **2018**, *51*, 1–37. [CrossRef]
18. Visual Studio IntelliCode. Available online: https://visualstudio.microsoft.com/services/intellicode/ (accessed on 30 March 2021).
19. Cao, H.; Meng, Y.; Shi, J.; Li, L.; Liao, T.; Zhao, C. A Survey on Automatic Bug Fixing. In Proceedings of the 2020 6th International Symposium on System and Software Reliability (ISSSR), Chengdu, China, 24–25 October 2020; pp. 122–131.
20. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, L.; Polosukhin, I. Attention is all you need. *arXiv* **2017**, arXiv:1706.03762.
21. Drain, D.; Wu, C.; Svyatkovskiy, A.; Sundaresan, N. Generating Bug-Fixes Using Pretrained Transformers. *arXiv* **2021**, arXiv:2104.07896.
22. Ueda, Y.; Ishio, T.; Ihara, A.; Matsumoto, K. Devreplay: Automatic repair with editable fix pattern. *arXiv* **2020**, arXiv:2005.11040.
23. Gupta, R.; Pal, S.; Kanade, A.; Shevade, S. Deepfix: Fixing common c language errors by deep learning. In Proceedings of the AAAI Conference on Artificial Intelligence, San Francisco, CA, USA, 4–9 February 2017.
24. Hajipour, H.; Bhattacharya, A.; Fritz, M. SampleFix: Learning to Correct Programs by Sampling Diverse Fixes. *arXiv* **2019**, arXiv:1906.10502.

25. Gupta, R.; Kanade, A.; Shevade, S. Neural Attribution for Semantic Bug-Localization in Student Programs. In Proceedings of the 2019 Conference on Advances in Neural Information Processing Systems, Vancouver, BC, Canada, 8–14 December 2019; pp. 11861–11871.
26. Vasic, M.; Kanade, A.; Maniatis, P.; Bieber, D.; Singh, R. Neural program repair by jointly learning to localize and repair. *arXiv* **2019**, arXiv:1904.01720.
27. Wong, W.E.; Gao, R.; Li, Y.; Abreu, R.; Wotawa, F. A survey on software fault localization. *IEEE Trans. Softw. Eng.* **2016**, *42*, 707–740. [CrossRef]
28. Lee, J.; Song, D.; So, S.; Oh, H. Automatic diagnosis and correction of logical errors for functional programming assignments. *Proc. ACM Program. Lang.* **2018**, *2*, 1–30. [CrossRef]
29. Matsumoto, T.; Watanobe, Y. Towards hybrid intelligence for logic error detection. In *Advancing Technology Industrialization through Intelligent Software Methodologies, Tools and Techniques, Proceedings of the 18th International Conference on New Trends in Intelligent Software Methodologies, Tools and Techniques (SoMeT_19), Kuching, Malaysia, 23–25 September 2019*; IOS Press: Amsterdam, The Netherlands, 2019; Volume 318, p. 120.
30. Matsumoto, T.; Watanobe, Y. Logic Error Detection Algorithm Based on RNN with Threshold Selection. In *Knowledge Innovation through Intelligent Software Methodologies, Tools and Techniques, Proceedings of the 19th International Conference on New Trends in Intelligent Software Methodologies, Tools and Techniques (SoMeT_20), Kitakyushu, Japan, 22–24 September 2020*; IOS Press: Amsterdam, The Netherlands, 2020; Volume 327, p. 76.
31. Hochreiter, S.; Schmidhuber, J. Long short-term memory. *Neural Comput.* **1997**, *9*, 1735–1780. [CrossRef]
32. Bengio, Y.; Ducharme, R.; Vincent, P.; Jauvin, C. A neural probabilistic language model. *J. Mach. Learn. Res.* **2003**, *3*, 1137–1155.
33. Vapnik, V. Pattern recognition using generalized portrait method. *Autom. Remote Control* **1963**, *24*, 774–780.
34. Watanobe, Y. Development and Operation of an Online Judge System. *IPSJ Mag.* **2015**, *56*, 998–1005.
35. Aizu Online Judge. Developers Site (API). Available online: http://developers.u-aizu.ac.jp/index (accessed on 30 March 2021).
36. Hotelling, H. The Generalization of Student's Ratio. *Ann. Math. Stat.* **1931**, *2*, 360–378. [CrossRef]
37. Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G.S.; Davis, A.; Dean, J.; Devin, M.; et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015. Available online: https://www.tensorflow.org/ (accessed on 30 March 2021).
38. Kingma, D.P.; Ba, J. Adam: A method for stochastic optimization. *arXiv* **2014**, arXiv:1412.6980.
39. Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I.; Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* **2014**, *15*, 1929–1958.
40. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; et al. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* **2011**, *12*, 2825–2830.
41. Yi, J.; Ahmed, U.Z.; Karkare, A.; Tan, S.H.; Roychoudhury, A. A feasibility study of using automated program repair for introductory programming assignments. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017; pp. 740–751.