applied sciences

MDPI

Article

Source Code Assessment and Classification Based on Estimated Error Probability Using Attentive LSTM Language Model and Its Application in Programming Education

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; keita-n@u-aizu.ac.jp

* Correspondence: mostafiz26@gmail.com (M.M.R.); yutaka@u-aizu.ac.jp (Y.W.)

Received: 23 March 2020; Accepted: 21 April 2020; Published: 24 April 2020

Abstract: The rate of software development has increased dramatically. Conventional compilers cannot assess and detect all source code errors. Software may thus contain errors, negatively affecting end-users. It is also difficult to assess and detect source code logic errors using traditional compilers, resulting in software that contains errors. A method that utilizes artificial intelligence for assessing and detecting errors and classifying source code as correct (error-free) or incorrect is thus required. Here, we propose a sequential language model that uses an attention-mechanism-based long short-term memory (LSTM) neural network to assess and classify source code based on the estimated error probability. The attentive mechanism enhances the accuracy of the proposed language model for error assessment and classification. We trained the proposed model using correct source code and then evaluated its performance. The experimental results show that the proposed model has logic and syntax error detection accuracies of 92.2% and 94.8%, respectively, outperforming state-of-the-art models. We also applied the proposed model to the classification of source code with logic and syntax errors. The average precision, recall, and F-measure values for such classification are much better than those of benchmark models. To strengthen the proposed model, we combined the attention mechanism with LSTM to enhance the results of error assessment and detection as well as source code classification. Finally, our proposed model can be effective in programming education and software engineering by improving code writing, debugging, error-correction, and reasoning.

Keywords: language modeling; classification; error probability; error assessment; logic error; neural network; LSTM; attention mechanism; programming education

1. Introduction

A huge amount of software is written in educational institutions and industry, making software reliability increasingly important. Source code usually contains multiple types of error, including syntax, semantic, communication, calculation, and logic errors. A single error is often enough to cause software failure. It is sometimes difficult for student or professional programmers to identify logic errors in source code, even with the help of traditional compilers. Helping programmers, especially novice programmers, properly assess and classify source code errors has become an important research topic in software engineering and programming education [1,2]. In general, software is debugged before it is released. Each software package must pass several testing phases. A crucial testing phase is error debugging. Student and professional programmers spend a huge amount of time trying to find source code errors. The entire source code must be searched to find even a single error, which is a tedious, cumbersome, and time-consuming task. Student and professional programmers often make

some common errors, such as missing semicolons, delimiters, or braces, and logic errors. These errors may be caused by a lack of experience or attention to detail. Both novice and experienced programmers make such errors, as reported in a study of programmers who build errors (Google) [3].

Machine learning (ML)-based classifiers can predict source code errors after being trained on a correct source code corpus [4–6]. Source code classifiers can assist programmers in fixing potential errors, thereby increasing source code correctness and reliability. Traditional source code error prediction methods consist of two steps, namely the extraction of features from training datasets and the development of an ML model (supervised or unsupervised) for classification. Previous research has concentrated on the design of preferential metrics to obtain higher accuracy. Features can be divided into Halstead [7] features depending on operators and operands, McCabe [8] features, and CK [9] features extracted from object-oriented programs. Most supervised and unsupervised classifiers are unable to properly classify source code using extracted features, inside the features logic, syntax, and semantic errors may exist. Feature-based traditional classifiers consider only the current features instead of checking all source code sequences.

Due to the sensitivity of source code, error assessment, detection, and classification is a challenging task. Traditional compilers cannot accurately assess source code errors. Therefore, a method based on artificial intelligence (AI) is required to assist programmers in the assessment and detection of such errors. Artificial neural networks (ANNs) are attractive for this task.

Natural language processing (NLP) has recently produced a lot of remarkable results in applications such as language processing, speech recognition, and machine translation. An n-gram model is an example of a stochastic language model for predicting the next item or word based on a large text corpus. N-gram models such as bi-gram, tri-gram, skip-gram [10], and GloVe [11] are statistical language models that can be applied to language modeling. The availability of large text corpuses has made NLP techniques effective. A language model is useful and intuitive for short repeated source code sequences. However, for complex software engineering, the NLP language model is less useful. Many researchers have focused on source code error assessment and classification using language modeling. An ANN-based language model could be a replacement for error assessment and detection as well as source code classification. Recurrent neural network (RNN)-based models have recently achieved some success in language modeling. An RNN can hold a larger source code sequence context compared with that for traditional n-gram and other language models [12]. RNNs have limitations in terms of representing such large contexts due to gradient vanishing or exploding [13], making it difficult to train RNN-based models using long source code sequences. RNNs are thus effective for only short source code sequences. RNNs have been extended to long short-term memory (LSTM) networks to avoid gradient vanishing or exploding. LSTM can remember both short and long source code sequences using an internal gate structure.

In this paper, we present a language model for assessing and detecting various source code errors (logic, syntax, semantic, runtime, etc.) as well as classifying the source code as correct (error-free) or incorrect based on the estimated error probability. We developed the language model using LSTM combined with the attention mechanism (hereafter referred to as LSTM-AttM). LSTM-AttM is more powerful and effective than a basic RNN, standard LSTM, and other traditional baseline models. We trained RNN, LSTM, and LSTM-AttM models with various numbers of hidden layers (50, 100, 200, 300, and 400) using a large correct source code corpus collected from an online judge system. For the evaluation process, source code with and without errors were used as the input to the model. The model then assessed and detected syntax and logic errors with locations in code and classified the source code as either correct or incorrect based on the estimated error probability. The LSTM-AttM model can detect many common errors in source code, including logic errors. The LSTM-AttM network can use long source code sequences as the input to generate the optimal output. The proposed model was tuned with various numbers of hyperparameters and hidden layers to optimize it in terms of perplexity, accuracy, training time, and other performance measurement metrics. The output of the

proposed model will be helpful for student and professional programmers as well as programming education and software engineering. The contributions of our research are as follows:

- Our proposed model can provide a thorough evaluation of source code which includes error detection, correct word prediction with line numbers, as well as classification. Thus, for learning programming, the model can act as an intelligent compiler.
- The logic and syntax error detection accuracies are 92.2% and 94.8%, respectively, which are much better than those for state-of-the-art models.
- The proposed model can classify source code as being either correct or incorrect based on the estimated error probability. The average precision, recall, and F-measure values for source code classification based on syntax and logic errors are much higher than those of reference benchmark models.
- We combined the attention mechanism with the proposed neural network model to strengthen the language model. Generally, in source codes, a single line can have a long dependency on the previous line, in which case the attention mechanism uses all the hidden states of the past to make accurate predictions.
- The proposed model can help novice and experienced programmers quickly fix their source code, thus saving valuable time.

The rest of this paper is organized as follows. Section 2 presents the background and literature review. Section 3 describes LSTM neural networks. Section 4 presents the proposed approach. Section 5 presents the data collection and normalization processes. Section 6 presents the experimental results and evaluations. Section 7 discusses the results. Finally, Section 8 concludes this research and provides suggestions for future work.

2. Background and Literature Review

In the source code, a single line may have reliance on the preceding lines, making it difficult to evaluate complex source code by any conventional language model. The LSTM based language model is a promising method for source code error assessment and classification.

Information and communication technology has become an influential economic catalyst. A huge amount of source code is written and compiled globally. AI can be applied to assess source code errors. AI-based language models are often used for source code assessment and classification to obtain human-like responses. Many researchers have used AI-based models to detect source code errors in software engineering and programming education.

Pu et al. [10] proposed a source code correction method based on LSTM using code segment similarities. The study leveraged the sequence-to-sequence (seq2seq) neural network model with natural language processing tasks for the code correction process. Another study [12] proposed a deep software language model based on RNNs. The experimental results showed that the model outperforms traditional language models such as n-gram and cache-based n-gram in a Java corpus. The software language model shows great promise in the field of software engineering. Terada et al. [14] proposed an LSTM-based model for programming education where the model predicts the next word by analyzing incomplete source code. Novice programmers often struggle to write a complete program from scratch. To help them, the model predicts the next word to complete a program. The LSTM-based model achieved a high degree of prediction accuracy. Fault detection in source code has become an important research topic [1]. In one study [15], source code defect prediction was performed based on churn metrics combined with source code dependencies. In another [16], an extensive analysis of metrics and static code attributes was conducted for error prediction. Arar et al. [17] selected suitable features by employing a naive Bayes classifier. Jing et al. [18] introduced a vocabulary learning model that calculates the incorrect classification cost for the prediction of source code defects. Various ML approaches [19–21] have been proposed for classification, recommendation, and estimation problems. Alreshedy et al. [22] presented an ML-based language model for classifying source code

snippets based on the programming language. In their work, a multinomial naive Bayes classifier was applied and code snippets from the website Stack Overflow were used as experimental data. Ram and Nagappan [23] proposed a hierarchical model that uses convolutional neural networks (CNNs) and LSTM for sentiment analysis in software engineering. This analysis model outperforms reference state-of-the-art models. Reyes et al. [24] classified archived source code by type of programming language using an LSTM network. Empirical results showed that the LSTM network outperformed the naive Bayes classifier and linguist classifier.

Terada and Watanobe [25] presented a method for the automatic generation of fill-in-the-blank problems for novice programmers using k-means clustering and the bidirectional LSTM model. The k-means clustering method is used to select ideal source code from an online judge system and the code to be made blank (to be filled with appropriate words using the bidirectional LSTM model). Tai et al. [26] presented a model called Tree-LSTM where an LSTM network works like a tree. The model evaluates the tasks of prediction of semantic relatedness based on sentence pairs and sentiment classification. Pedroni and Meyer [27] presented a survey-based analysis that focused on what type of compiler message helps novice programmers identify errors and what actions should be taken regarding source code errors. They experimentally showed which type of message helps most. Saito and Watanobe [28] proposed a learning path recommendation system for novice students based on their desired learning ability chart. The students were clustered and an excellent student from each cluster was selected. The model extracted features from the selected excellent students. Finally, the model used the features as training input to the neural network. An LSTM network was used to predict the learning path of the students. In another study [29], a source code bug detection technique that uses LSTM was proposed. The hyperparameters of LSTM were adjusted to determine the optimal perplexity and training time. The LSTM network produces a plausible outcome for source code bug detection. Fan et al. [30] presented an attention-based RNN for source code defect prediction. F-measure score and the area under the curve (AUC) were used as model evaluation metrics. The proposed model improved the source code classification process. The F-measure score and AUC had 14% and 7% better accuracy than those of state-of-the-art models, respectively. Ohashi et al. [31] proposed a source code classification model that uses a CNN. The model classifies source code based on the type of algorithm in the code. During CNN model training, all source code is converted into a simple structure of code without any variables, functions, keywords, etc. The obtained classification accuracy of the CNN model is very high.

In summary, many promising methods have proposed. Most researchers utilized traditional supervised and unsupervised classifiers, RNNs, LSTM, or CNNs as language models for source code classification and other applications. RNNs are much better than traditional language models such as n-gram, but have limitations in terms of handling long input sequences. LSTM is a variant of RNNs that overcomes the shortcomings of RNNs. The model proposed in the present study combines the attention mechanism with LSTM (LSTM-AttM). The LSTM-AttM network is used as a language model for source code assessment and classification based on the estimated error probability. The LSTM-AttM network outperforms LSTM because the latter uses only the last hidden state outcome for prediction. In contrast, LSTM-AttM considers all previous hidden state outcomes for prediction. Most of the studies used different models for source code classification based on errors, programming language detection, archive code classification, and simple error detection. On the other hand, our proposed model specifically identifies logic, syntax and other errors in the source code. Furthermore, the model can predict the correct words in place of the error location. Overall our proposed LSTM-ATM model differs from other models in achieving unique goals.

3. Long Short-Term Memory Network

An LSTM network is a type of RNN. The LSTM network has been effectively used in the field of deep learning. The main advantage of an LSTM network is ease of training because it does not face problems such as gradient vanishing or exploding. LSTM can process entire input (source code, video,

speech, image) sequences. An LSTM network memory unit consists of four attributes, namely a forget gate, a cell state, an input gate, and an output gate. The cell state remembers the information of the entire sequence and the three gates control the input and output of the cell, as shown in Figure 1.

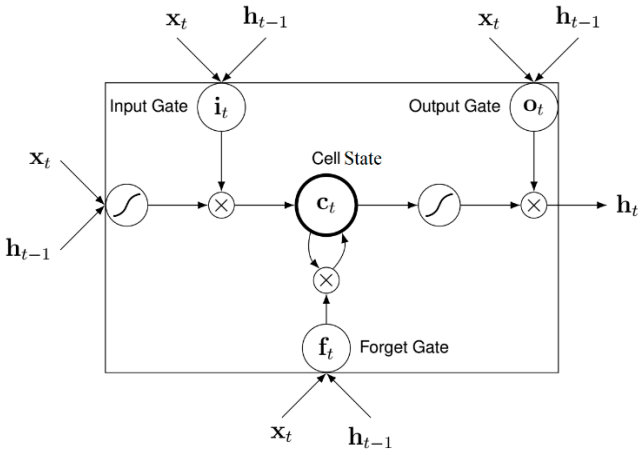


Figure 1. Internal structure of a simple long short-term memory (LSTM) unit.

At the start of the process, the forget gate checks which information to throw away and which information to keep in the cell state. Equation (1) is used for the forget gate. It is calculated at cell state c_{t-1} using hidden state h_{t-1} and input x_t. The output of the forget gate, between 0 and 1, is produced by the sigmoid function. An output value of 1 (0) means keep (remove) all information in (from) the cell state.

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

To store a new piece of information in the cell state, the input gate decides which value will be updated using the sigmoid function. The tanh function creates a new candidate value $\widetilde{c_t}$ for the cell state.

$$i_t = \sigma(w_i \,.\, [h_{t-1}, x_t] + b_i) \tag{2}$$

$$\widetilde{c_t} = \tanh(w_c \,.\, [h_{t-1}, x_t] + b_c) \tag{3}$$

Then, the old cell state c_{t-1} is used to update c_t.

$$c_t = f_t \,*\, c_{t-1} + i_t \,*\, \widetilde{c_t} \tag{4}$$

We can now calculate the output of LSTM, which is based on a filtered version of the cell state. The sigmoid function decides which part of the cell state is going to the output and then updates the weight accordingly.

$$o_t = \sigma(w_o \,.\, [h_{t-1}, x_t] + b_o) \tag{5}$$

$$h_t = o_t \,*\tan \mathrm{h}(c_t) \tag{6}$$

The combination of the attention mechanism with LSTM improves model performance for fault assessment and detection and the classification of source code.

4. Proposed Approach

In the proposed model, an LSTM-AttM network is used as a seq2seq language model for error assessment and detection as well as source code classification. We trained the proposed model using correct source code. The model then generated the error probability through the softmax layer for each error candidate word based on the context vector c_t of all previous hidden states and the current state output h_t. The estimated error probability is also used to classify the source code as either correct

(error-free) or incorrect. The proposed LSTM-AttM model can identify many kinds of error (logic, syntax, semantic, etc.) in source code to increase source code reliability. The workflow of the proposed model is shown in Figure 2.

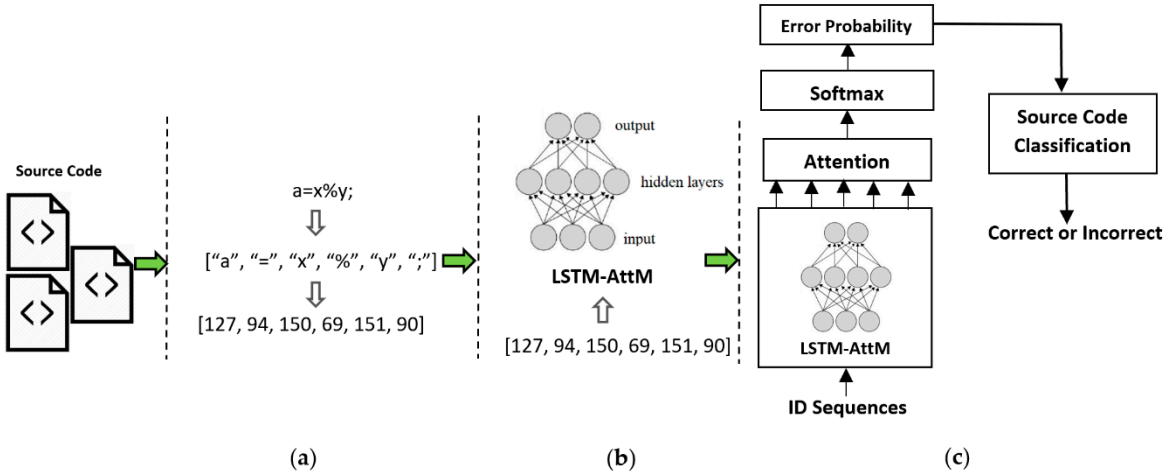


Figure 2. Workflow of proposed model: (**a**) word embedding and encoding process, (**b**) training of LSTM combined with the attention mechanism (LSTM-AttM) network using IDs, and (**c**) error probability prediction followed by the softmax layer and source code classification.

Proposed LSTM-AttM Model Architecture

The attention mechanism has been adapted for performing various tasks [32–35]. It is most commonly used in seq2seq modeling. A neural network that utilizes the attention mechanism is called an attentive neural network. The conventional seq2seq model cannot properly process a long sequence of input because only the last hidden state of the input is used as a context vector for output [36]. The attention mechanism maps the most relevant words from the input sequence and then assigns a higher weight to these words to enhance the output accuracy. We incorporated the attention mechanism with LSTM, as shown in Figure 3, to better predict short and long sequences of source code. The proposed LSTM-AttM model creates a potential application domain in programming education arena.

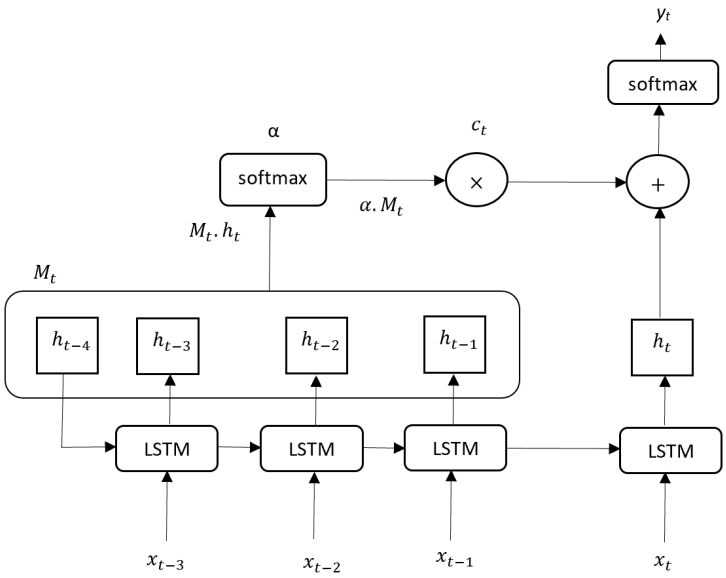


Figure 3. Architecture of proposed LSTM-AttM network model.

Attention is a vector or dense output layer with a softmax function. It is used to enhance the performance of machine translators and seq2seq models. Recently, the attention mechanism has achieved great success in machine translation tasks. A machine translator sometimes compresses long sequenced sentences into a fixed-length vector. Therefore, information may be lost. The attention mechanism mitigates this problem. Although LSTM has outstanding performance in terms of capturing long-range dependencies, a hidden state carries all the information into a fixed-length vector [36]. The attention mechanism has been applied to neural language models such as LSTM to overcome this problem [37]. The attention mechanism allows a neural language model to retrieve and make use of pertinent information in all previous hidden states, improving network retention. The mathematical details of the attention mechanism are described in previous work [38]. For attention, we use external memory M for previous hidden states, which is denoted as $M_t = [h_{t-M} \ldots\ldots\ldots\ldots h_{t-1}] \epsilon \mathbb{R}^{k*M}$. At time step t, the context vector c_t and attention weight α_t. Now, the model uses the attention layer between h_t and the hidden states in M_t. We defined our attention-based LSTM model by the following equations.

$$A_t = M_t.h_t \tag{7}$$

$$\alpha_t = softmax(A_t) \tag{8}$$

$$c_t = M_t \alpha_t^T \tag{9}$$

For predicting the next word at time step t, the calculation is based on current hidden states h_t and context vector c_t. The vocabulary spaces are obtained using the softmax function to produce the final probability $y_t \epsilon \mathbb{R}^v$. G_t is an output vector.

$$G_t = \tan h\Big(w^g\Big[w^h(h_t) + w^m(c_t)\Big]\Big) \tag{10}$$

$$y_t = softmax(w^v G_t + b^v) \tag{11}$$

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 trainable bias vector, and v is the vocabulary size.

The attention mechanism facilitates the extraction of more accurate features from input sequences, and thus the LSTM-AttM network increases the performance of the proposed model.

5. Data Collection and Normalization

In the present research, we collected all the datasets from the Aizu Online Judge (AOJ) system [39,40]. The AOJ system has more than 2000 problems and 65,000 users as of February 2020. The problems and algorithms are divided into categories [28]. The AOJ system has more than 4 million source code samples for various problems. A total of 18 programming languages, including C++, C, Ruby, and Python, are supported by the AOJ system. The system keeps all statistical information on programming and the submission logs of individual users. These resources can be used to conduct research in programming education and software engineering. To train the proposed model, we took correct solutions for Insertion Sort (IS), Greatest Common Divisor (GCD), Prime Numbers (PN), Bubble Sort (BS), and Selection Sort (SS) problems from the AOJ system. All the source code was written in the C language. The selected source code was archived on the AOJ system from August 2018 to September 2019. The total numbers of correct source code submissions for IS, GCD, PN, BS, and SS are 2285, 1821, 1538, 2425, and 2294, respectively. The overall solution success rates for IS, GCD, PN, BS, and SS are 35.16%, 49.86%, 30.8%, 47.74%, and 59.79%, respectively. A total of 10,362 correct and incorrect source codes were used for model training where the number of correct and incorrect codes was equal. Of the total source codes, we used 90% of the code for model training and 10% for testing. To evaluate the error in the source code, we randomly selected 100 new source codes from each category. A total of 500 source codes were examined by the model for logical, syntax, and others

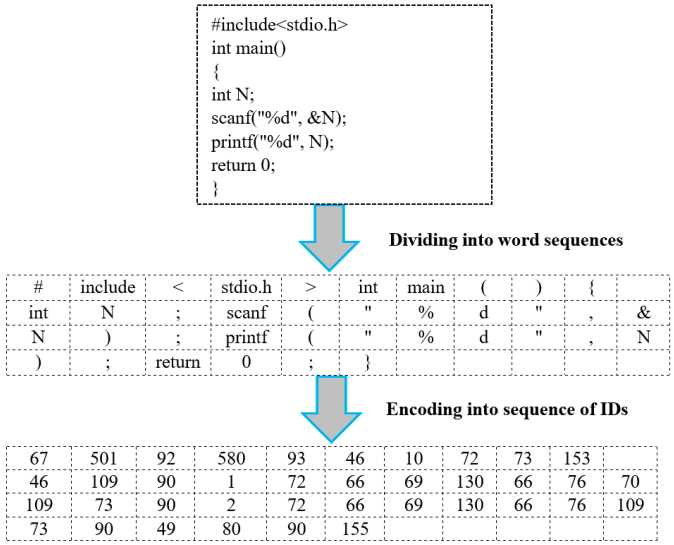
error evaluation. For classification, we selected approximately 1300 erroneous source codes from all categories to evaluate the effectiveness of the classification.

Before model training, we refined all source code by removing unnecessary elements. We adopted the source code conversion procedure applied in a previous study [29]. Initially, we removed all comments, line breaks (\n), and tabs (\t) from the source code because they are not relevant for error assessment and classification. The source code was converted to word sequences and then functions, keywords, variables, and characters were considered as normal words. Each word was encoded with an ID. The IDs for functions, variables, keywords, and characters are shown in Table 1. Any user-defined functions and variables in the source code not defined in Table 1 were assigned unique IDs from a defined range in the encoding process. The entire process, called word embedding and encoding, is shown in Figure 4.

Table 1. Partial list of defined IDs for keywords, characters, and numbers.

ID	Word	ID	Word	ID	Word	ID	Word
30	auto	46	int	62		78	.
31	break	47	long	63	!	79	/
32	case	48	register	64	?	80	0
33	char	49	return	65	_	81	1
34	const	50	short	66	“	82	2
35	continue	51	signed	67	#	83	3
36	default	52	sizeof	68	$	84	4
37	do	53	static	69	%	85	5
38	double	54	struct	70	&	86	6
39	else	55	switch	71	‘	87	7
40	enum	56	typedef	72	(	88	8
41	exturn	57	union	73	)	89	9
42	float	58	unsigned	74	*	90	;
43	for	59	void	75	+	91	:
44	goto	60	volatile	76	,	92	<
45	if	61	while	77	~	93	>
94	=	110	O	126	‘	142	p
95	@	111	P	127	a	143	q
96	A	112	Q	128	b	144	r
97	B	113	R	129	c	145	s
98	C	114	S	130	d	146	t
99	D	115	T	131	e	147	u
100	E	116	U	132	f	148	v
101	F	117	V	133	g	149	w
102	G	118	W	134	h	150	x
103	H	119	X	135	i	151	y
104	I	120	Y	136	j	152	z
105	J	121	Z	137	k	153	{
106	K	122	[	138	l	154	|
107	L	123	\	139	m	155	}
108	M	124	]	140	n		
109	N	125	^	141	o		

After the training process, the performance of the model was evaluated in terms of source code assessment and classification accuracy. To predict the next ID sequence, the model uses the prefix of all ID sequences using the attention mechanism. The ID sequences are transformed in several phases followed by a softmax layer to generate the probability for the next ID sequence or candidate word. In the proposed model, a word is considered as an error candidate whose probability is less than 0.1 [29]. The difference between the predicted and actual results is called perplexity. The perplexity is calculated at the softmax layer at each time step to observe the loss function.



#	include	<	stdio.h	>	int	main	(	)	{	
int	N	;	scanf	(	"	%	d	"	,	&
N	)	;	printf	(	"	%	d	"	,	N
)	;	return	0	;	}					

67	501	92	580	93	46	10	72	73	153	
46	109	90	1	72	66	69	130	66	76	70
109	73	90	2	72	66	69	130	66	76	109
73	90	49	80	90	155					

Figure 4. Word embedding and encoding process for source code.

The softmax layer receives the vector $x = [x_1, x_2, \ldots \ldots \ldots ., x_n]$ and returns the probability vector $p = [\, p_1, p_2, p_3, \ldots \ldots .., p_n]$, expressed as follows:

$$P_i = \frac{\exp(x_i)}{\sum_{j=1}^{k} \exp\left(x_j\right)} \tag{12}$$

where $i = 1, 2, 3, 4, \ldots \ldots ., k$.

Perplexity, expressed below, is a standard performance measurement. It indicates how well a probability model predicts a sample. A lower value indicates a better model.

$$H_p \approx -\frac{1}{N} \sum_{i}^{m} log_2 p \left(W_i | W_{i-n+1}^{i-1}\right) \tag{13}$$

where $|N|$ is the length of the sample, w_i is an ID in a sample, and $P(w_i)$ is the probability of w_i.

6. Experimental Results and Evaluations

We developed a general model that can be trained on any type of problem set. In the present research, we selected the source code for IS, PN, GCD, BS, and SS for the experiments. We trained an RNN, LSTM, and the LSTM-AttM network with various numbers of hidden layers (50, 100, 150, and 200). We recorded the epoch-wise perplexity and training times during the training period. The perplexity determines the efficacy of a language model. The evaluation and training processes were performed on a computer with an Intel Core i7-5600U CPU (2.60 GHz) with 8 GB of RAM running 64-bit Windows 10.

6.1. Experimental Setup

In our study, we use Python's chainer framework to create deep learning model architecture. Also, we considered the large number, length, and complexity of the source code to develop our proposed model. Before the start of training, we defined several hyperparameters for the experiment to obtain better results. First, we determine the number of hidden layers and epochs. Then the number of neurons was determined based on the number of hidden layers. Thus, the neurons were equal to the defined number of hidden layers. For example, If hidden layers $h_l = 100, 200, 300, 400$, and so on. Thus, the neurons at each hidden layer will be equal to the number of hidden layers, such as $n\_units = h_1$

where *n_units* = neurons at each layer and h_l = number of hidden layers. Dropout was used to regularize the LSTM network performance to avoid overfitting. To obtain better training accuracy dropout ratio was set to 0.5 [41]. We optimized the LSTM network using the Adam optimization algorithm [42]. Particularly, optimizer smoothing the model learning by binding together loss function and model parameters in order to produce better training accuracy. The learning rate or step size of our network was $l = 0.001$. The network weights were updated based on the value of l during training. A higher (lower) value of l makes initial learning faster (slower). The values of β_1 and β_2, the exponential decay rates for the first- and second-moment estimates, were set to 0.001 and 0.999, respectively. It is often effective to reduce the learning rate when training is running. Without exponential decay, the loss function cannot start again to diverge after decreased a certain point. The value of ε ($= 1e^{-8}$) was used to prevent division by zero in the implementation. We trained our network with various numbers of hidden layers (50, 100, 150, 200, 250, 300, and 400). The corresponding models are called the 50-layer model, 100-layer model, and so on. We evaluated the performance of all models to determine the optimal number of hidden layers.

6.2. Perplexity, Training, and Hidden Layer Selection

The performance of a language model strongly depends on training time and perplexity. Perplexity also determines how good a model training process as well as calculates the model loss function. During training with various numbers of hidden layers, we calculated the epoch-wise perplexity to determine the optimal number of hidden layers. Correct source code samples were selected from the AOJ system for training. The perplexity at the last epoch (30th) of training for each type of program is shown in Figure 5.

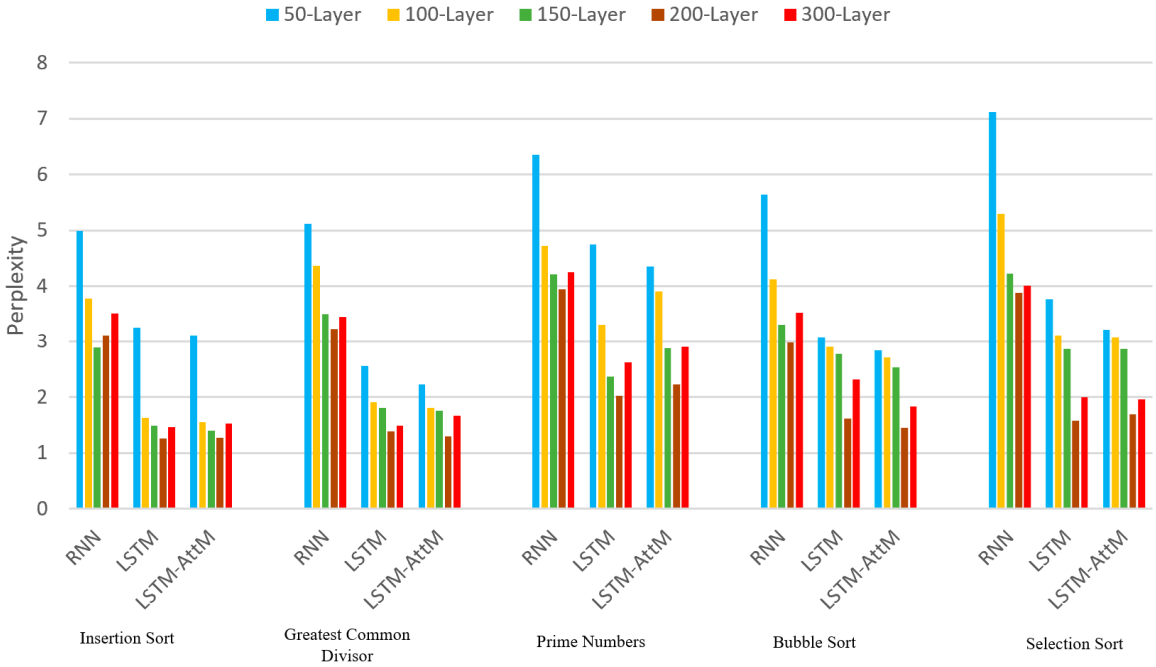


Figure 5. Perplexity values at last epoch of proposed and other state-of-the-art models during the training period for various problem sets.

The figure shows that the 200-layer model had the lowest perplexity during the training period. The epoch-wise perplexity for the 200-layer model for various problem sets is shown in Figure 6.

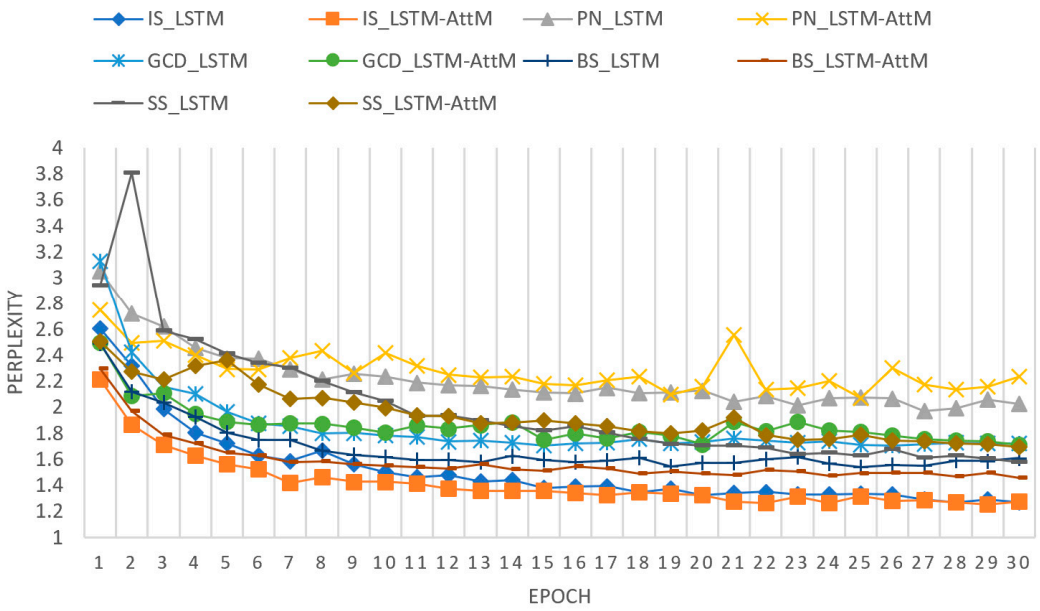


Figure 6. Epoch-wise perplexity for LSTM and LSTM-AttM models with 200 hidden layers for various problems.

Based on these results, we selected 200 hidden layers for LSTM-AttM and the other state-of-the-art models for all subsequent experiments. The training accuracies for the RNN, LSTM, and LSTM-AttM models are listed in Table 2.

Table 2. Average training accuracy of models for various problem sets.

Problem	Training Accuracy (%)					
	Correct Source Codes			Incorrect Source Codes		
	RNN	LSTM	LSTM-AttM	RNN	LSTM	LSTM-AttM
Insertion Sort	70	81	94	68.3	82.4	93.6
Greatest Common Divisor	68	80	92	71.5	81	92.3
Prime Numbers	75	83	90	73.5	84.5	90
Bubble Sort	72	79	89	73.2	81.7	90
Selection Sort	65	78	87	66	80	89.6

After model training, we evaluated the performance of the proposed model in terms of the detection of syntax, logic, and other errors as well as source code classification (correct or incorrect). We selected source code with errors for model validation and testing. Our goal was to evaluate the performance of the proposed model in terms of how accurately it assesses and detects errors in source code. To evaluate model performance, we adopted three evaluation indices, namely error detection accuracy (*EDA*), error prediction accuracy (*EPA*), and model accuracy (*MA*), respectively defined below.

$$EDA = \frac{Actual\ Error\ Word\ (AEW)}{Total\ Detected\ Errors\ (TDE)} \times 100\% \tag{14}$$

$$EPA = \frac{Actual\ Correct\ Word\ (ACW)}{Total\ Predicted\ Words\ (TPW)} \times 100\% \tag{15}$$

$$MA = \frac{EDA + EPA}{2} \tag{16}$$

The proposed model detects errors in source code by utilizing the trained correct source code corpus. Of the detected errors, there are some true errors, which are called actual error words (*AEW*s). Of the predicted words, there are some true correct words, which are called actual correct words

(*ACW*s). It is noted that the estimated probabilities of *AEW* and *ACW* should be more than 0.90. We used the above-mentioned evaluation indices to measure the performance of the models in terms of syntax and logic error assessment and detection.

6.3. Syntax Error Assessment and Detection

A syntax error is an error where the program violates a structural rule of a certain programming language. To compile, source code must follow the structural rules of a programming language, if it does not, the compiler will output syntax errors. Common examples of syntax error include misspelled keywords, missing single or double quotes, missing matching brackets, and a missing semicolon at the end of a statement. To assess and detect syntax errors in source code, the proposed LSTM-AttM language model calculates the error probability of each error candidate word. The error probability determines the possibility of syntax errors in source code. The proposed model assesses the source code thoroughly and detects syntax error candidates, as shown in Figure 7.

```
1 #include <stdio.h>
2
3 int main()
4 {
5   int i,j,n,key;
6   int A[100];
7
8   scanf("%d",n);
9   for (i=0;i<n;i++){
10    scanf("%d",&A[i]);
11  }
12
13  for (j=1;j<n;j++){
14    for (i=0;i<n-1;i++){
15      printf("%d ",A[i]);
17    printf("%d\n",A[n-1]);
18    key=A[j];
19    i=j-1;
20  while ((i>=0) & A[i]>key)){
21      A[i+1]=A[i];
22      i=i-0;
23    }
24    A[i+1]=key;
25  }
26  for (i=0;i<n-1;i++){
27    printf("%d ",A[i]);
28  }
29  printf("%d\n",A[n-1]);
30  return 0;
```

Figure 7. Syntax error assessment and detection for source code evaluated using LSTM-AttM.

In the figure, the proposed LSTM-AttM model assesses the source code and estimates the error probability for each error candidate word. The corresponding location (line number) of each detected word is listed in Table 3. The error probability determines the syntax error possibility for a particular candidate word and location in the source code. Although the model detected all the potential locations of syntax error, the detected error candidates might not have all been accurately identified. Words with an error probability of more than 0.98 are outlined in blue in Figure 7. We considered these errors to be confirmed syntax errors.

Table 3. Estimated error probability for source code in Figure 7.

Line Number	Error Candidate (Probability < 0.1)	Suggested Word	Estimated Error Probability
8	n	&	**0.9999918**
13	1	0	0.60096426
17	n	key	0.7747942
19	i	n	0.4699676
20	(	&	**0.98158526**
22	0	1	**0.9885606**

To compare our model with baseline models, in addition to the above-mentioned example (Figure 7) a large number of erroneous source code samples were used for the evaluation process.

The obtained results are listed in Table 4. The syntax error assessment and detection accuracy results for the proposed model for all problem sets are better than those for the state-of-the-art models.

Table 4. Assessment results of syntax error detection for erroneous source code.

Problem	Accuracy (%)		
	RNN	**LSTM**	**LSTM-AttM**
Insertion Sort	83	88	98
Greatest Common Divisor	81	90	95
Prime Numbers	74	85	93
Bubble Sort	80	80	96
Selection Sort	69	78	92
Average	**77.4**	**84.2**	**94.8**

6.4. Logic Error Assessment and Detection

A logic error in source code generates unexpected program output. The cause of logic error is typically the incorrect application of mathematical logic in source code. Conventional compilers cannot detect or assess logic error, and thus student and professional programmers must check the entire source code line by line. This is a major problem, especially for novice programmers. A simple program with logic error is shown in Figure 8. The program takes in an array of numbers and then outputs it. In the example, four numbers are given for an array but because of incorrect logic, only three of them are output.

```
#include<stdio.h>
int main()
{
int i,N, A[100];
scanf("%d", &N);
for(i=1; i<N; i++)
scanf("%d", &A[i]);
for(i=1; i<N; i++)
printf("%d ", A[i]);
return 0;
}
```

Input:
4
3 4 5 6
Output:
3 4 5

Figure 8. Example of source code with logic error and its input and output.

Logic error assessment and detection is a challenging task for traditional compilers. The proposed attention-based language model identifies logic error candidate words in source code to reduce the time required to check for such errors. To identify logic errors, the model should be able to calculate long dependent sequences of source code. We thus designed the seq2seq language model by combining the attention mechanism with LSTM. We compared its performance with other state-of-the-art models. Source code with logic error (an example for BS) was evaluated by the LSTM-AttM model. The results are shown in Figure 9. The source code assessment and detection results are listed in Table 5. The results reveal the effectiveness of the proposed LSTM-AttM model. The proposed model assessed and identified logic errors and their locations in source code. The estimated error probability ensures the logic error possibility on a particular line (blue outline) of source code. The model detected two logic errors on line 6 and generated the corresponding error probabilities (see Table 5). The estimated error probabilities are both more than 0.90, indicating possible logic errors on line 6 of the source code.

```
1 #include<stdio.h>
2 int main()
3 {
4  int a,b[100],i,j,n=0,m;
5  scanf("%d",&a);
6    for(i=1;i<a;i++){
7   scanf("%d",&b[i]);
8  }
9  for(i=0;i<a-1;i++){
10   for(j=a-1;j>i;j--){
11    if(b[j]<b[j-1]){
12    m=b[j];
13    b[j]=b[j-1];
14    b[j-1]=m;
15    n++;
16     }
17   }
18 }
19 for(i=0;i<a-1;i++){
20  printf("%d ",b[i]);
21 }
22 printf("%d\n",b[i]);
23 printf("%d\n",n);
24 return 0;
25 }
```

Figure 9. Logic error assessment and detection for source code evaluated using LSTM-AttM.

Table 5. Estimated error probability for erroneous source code in Figure 9.

Line Number	Error Candidates (Probability < 0.1)	Suggested Word	Estimated Error Probability
6	1	0	**0.9727551**
6	a	=	**0.92732173**

To assess logic errors, we selected source code from the AOJ system that generated a runtime error (i.e., failure during execution) judge verdict. Runtime errors can be caused by invalid pointer references (segmentation fault), overflow, division by zero, memory access violations, and uninitialized memory access. In the experiment, in addition to the above-mentioned example (Figure 9) a large number of source code samples with logic errors were used. The evaluation results are listed in Table 6. As shown, the proposed language model outperformed the reference benchmark models.

Table 6. Assessment results of logic error detection for erroneous source code.

Problem	Accuracy (%)		
	RNN	**LSTM**	**LSTM-AttM**
Insertion Sort	60	75	95
Greatest Common Divisor	57	81	96
Prime Numbers	63	77	90
Bubble Sort	65	80	91
Selection Sort	56	78	89
Average	**60.2**	**78.2**	**92.2**

6.5. Source Code Classification

In this section, we present the source code classification performance of the proposed LSTM-AttM and existing state-of-the-art models. We considered various kinds of error in source code, including semantic, syntax, logic, and communication errors. We evaluated the source code classification performance of the proposed model and state-of-the-art models by considering error occurrences in the source code. The proposed model calculated the error probability of each error candidate word to classify the source code.

In our model, each variable, keyword, operator, operand, class, function, etc. in the source code was considered as a normal word. The model generated the error probability for each error candidate

word followed by the softmax layer. In general, our model detects error candidate words and estimates the corresponding error probability for each one. If the estimated error probability for any word is greater than 0.90, the source code is classified as incorrect. To evaluate the classification performance, we compared our model with some baseline methods, namely standard LSTM, RNN, and the random forest (RF) method with a deep belief network (DBN) [43].

The performance of classification was evaluated in terms of precision, recall, and F-measure indices, respectively expressed as follows:

$$Precision\ (P_i) = \frac{TP_i}{TP_i + FP_i} \tag{17}$$

$$Recall\ (R_i) = \frac{TP_i}{TP_i + FN_i} \tag{18}$$

$$F - measure = \frac{2 * P_i * R_i}{P_i + R_i} \tag{19}$$

where TP_i is the true positive rate (erroneous source code classified as erroneous), FP_i is the false positive rate (correct source code classified as erroneous), and FN_i is the false negative rate (erroneous source code classified as correct). F-measure is the harmonic mean between precision and recall. Usually, it is difficult to always obtain excellent precision and recall. If all samples are classified as erroneous, the recall will be high but precision will be low. F-measure is a balance between precision and recall. The F-measure value is between 0 and 1, where a higher value indicates better classification.

We evaluated the model performance in terms of classification accuracy using source code samples with logic and syntax errors. Figure 10 shows the classification results for source code with syntax errors. The results show that the precision and recall values for the proposed model are better than those for the other models.

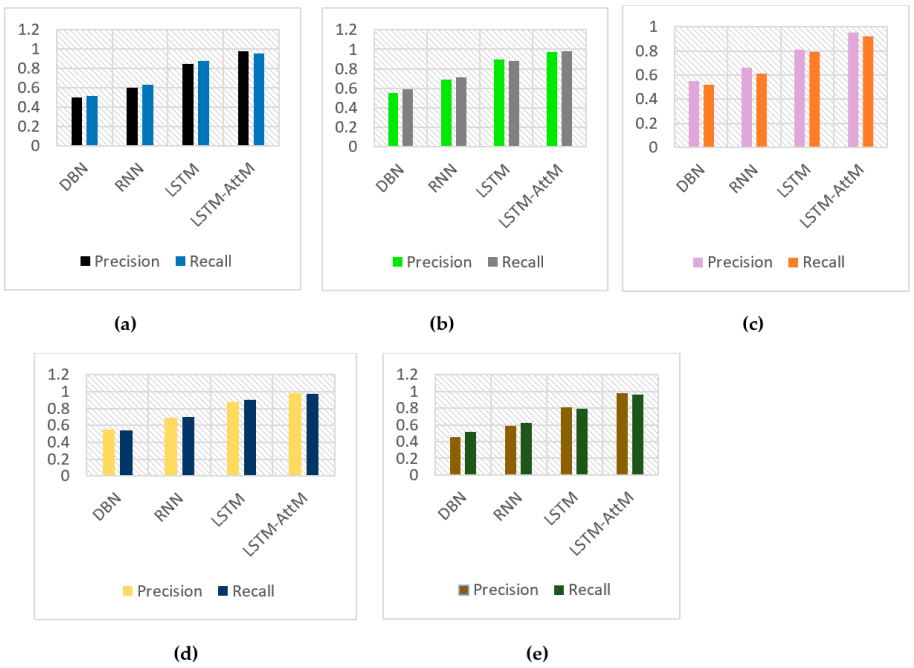


Figure 10. Comparison of precision and recall values for classification of source code with syntax errors for (**a**) Insertion Sort (IS), (**b**) Greatest Common Divisor (GCD), (**c**) Prime Numbers (PN), (**d**) Bubble Sort (BS), and (**e**) Selection Sort (SS).

We calculated the precision and recall values of each type of problem set. The proposed model had better values than those of the other models. The average precision, recall, and F-measure values are listed in Table 7.

Table 7. Average precision, recall, and F-measure values for classification of source code with syntax errors.

Model	Precision	Recall	F-Measure
DBN	0.50	0.50	0.50
RNN	0.54	0.58	0.56
LSTM	0.85	0.85	0.85
LSTM-AttM	0.97	0.96	**0.96**

Figure 11 shows the classification results for source code with logic errors. The average precision, recall, and F-measure values are listed in Table 8. The F-measure value indicates the excellent performance of the proposed LSTM-AttM model.

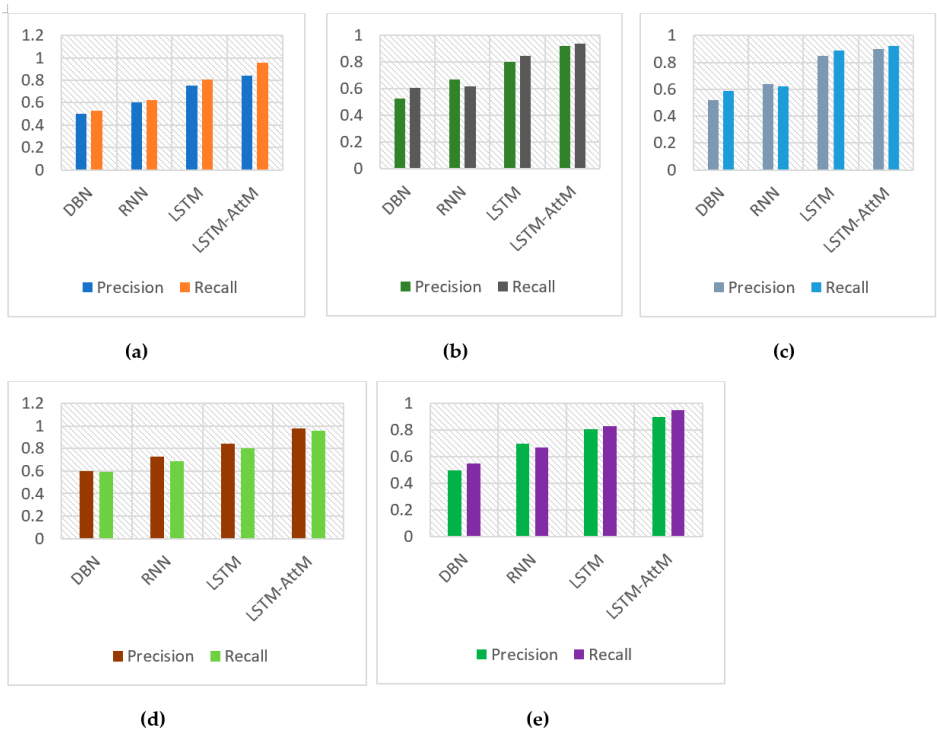


Figure 11. Comparison of precision and recall values for classification of source code with logic errors for (**a**) IS, (**b**) GCD, (**c**) PN, (**d**) BS, and (**e**) SS.

Table 8. Average precision, recall, and F-measure values for classification of source code with logic errors.

Model	Precision	Recall	F-Measure
DBN	0.53	0.50	0.51
RNN	0.55	0.56	0.55
LSTM	0.81	0.84	0.82
LSTM-AttM	0.91	0.95	**0.93**

6.6. Classification Result Comparison with Benchmark Models

We compared our experimental classification results with those for some baseline models. All the researches have a unique goal to achieve by respective research methodology. Nevertheless, we compared our proposed research with the most similar works. The results are presented in Table 9.

Table 9. Comparison with baseline models for defective source code classification.

Model	Description	F-Measure
RF+DBN [43]	RF is used for classification based on hidden features extracted using DBN.	0.50
RNN	A basic RNN is used to develop a language model for source code fault detection and prediction.	0.55
LSTM	A standard LSTM network is adapted to develop a language model for source code error detection, prediction, and classification.	0.79
DP-ARNN [30]	DP-ARNN is a defect prediction model that uses attention-based RNNs.	0.56
LSTM [29]	A source code error detection and prediction model based on a deep LSTM-based language model.	0.83
LSTM-AttM (Proposed)	A deep seq2seq language model that uses an attention mechanism + LSTM [29] network with customized hyperparameters for source code error assessment and detection and source code classification based on estimated error probability.	0.94

7. Discussion

The present research focused on source code fault assessment and classification. In software engineering and programming education, logic error assessment in source code is challenging for both student and professional programmers. We conducted experiments to assess and detect logic, semantic, and syntax errors in source code and classify source code as correct or incorrect using various models. The experimental results were compared with those for traditional unsupervised and other neural-network-based unsupervised models. The proposed model had the best performance.

The performance of a seq2seq language model strongly depends on the selection of the optimal number of hidden layers. This selection is based on the perplexity value. We calculated the perplexity during the training period. Figure 5 shows the perplexity of the RNN, LSTM, and LSTM-AttM models with various numbers of hidden layers at last epoch for various problems. The perplexity of 100, 150, and 300 layers are much higher than the 200 layers. Figure 6 shows the epoch-wise perplexity of models with 200 hidden layers for various problems. The perplexity was lowest for the 200-layer models that is why we selected 200-layer models for all experiments. We measured the training accuracy of the proposed language model and other models. The proposed LSTM-AttM model had the highest training accuracy (see Table 2).

In software engineering and programming education, the assessment and detection of logic errors in source code is a challenge. To address this problem, we used an attention-based language model using a deep LSTM neural network. After slight modification in source code pre-processing phase, the proposed model can be useful for any type of source code (Python, C++, Java, etc.). While a software system may be large, it has several functions (routines) that each have a limited number of lines. Some source codes are similar to such a routine. The difficulty level of each source code is not one, some source code uses complex mathematical logic and functions and some use simple. To evaluate our model, we used mixed (easy, medium, and hard) source code for error detection. The syntax error assessment and detection accuracy (see Table 4) for the proposed LSTM-AttM model was better than those for the LSTM and RNN models. The average accuracy of the proposed LSTM-AttM model was 94.8%, whereas those of LSTM and RNN models were 84.2% and 77.4%, respectively. The logic error assessment and detection accuracy is shown in Table 6. For logic error detection, the proposed model

(92.2%) outperformed LSTM (78.2%) and RNN (60.2%). To assess and detect logic errors in source code, the attention mechanism considers all input sequences because logic error detection is more complex than other error detection. The proposed model will especially help novice programmers most.

One of our main goals was to classify source code as either correct or incorrect. For this task, we used the estimated error probability of source code. The proposed LSTM-AttM model detects error candidates in source code and estimates the corresponding error probability for each error candidate word. The weight of the estimated error probability might vary because the language model generates the error probability for each error candidate word based on the training corpus. When the estimated error probability of any error candidate word is more than 0.90, the source code is treated as incorrect. The syntax error classification results are shown in Figure 10 and the average precision, recall, and F-measure values are listed in Table 7. The obtained precision, recall, and F-measure values of the proposed model are 0.97, 0.96, and 0.96, respectively, and those of the LSTM model are 0.85, 0.85, and 0.85, respectively. The precision and recall values for the classification of source code with logic errors for various problem sets are shown in Figure 11. The proposed model outperformed the LSTM and RNN models. The average precision, recall, and F-measure values for the classification of source code with logic errors are listed in Table 8. The average precision, recall, and F-measure values for the proposed model are 0.91, 0.95, and 0.93, respectively, better than those for the LSTM and RNN models. These classification comparison results verify the superiority of the proposed LSTM-AttM model over existing state-of-the-art models.

Source code classification results were also compared with those for some baseline models in Table 9. The F-measure value for the proposed model is 0.94, which is far better than those for the baseline models.

Finally, the experimental evaluation results demonstrate the superiority of the proposed model. Learners may get stuck when looking for logic errors and may thus spend a huge amount of time trying to fix them. In such cases, the proposed model can assist learners to accelerate the learning process. The model identifies errors and predicts the correct words, it also gives the line number for errors. The model can thus help students and programmers improve their programming skills and effectively create programs.

8. Conclusion and Future Work

In this study, we proposed an attention-based LSTM language model for assessing and classifying source code. In both programming education and software engineering, the proposed model can effectively help programmers. Conventional compilers cannot assess and detect logic errors in source code, and thus unexpected program output is generated. To avoid this adverse circumstance, the neural network-based language model achieves great success. The experimental results show that the accuracies of syntax and logic error detection using the LSTM-AttM model are approximately 94.8% and 92.2%, respectively. The proposed model calculates the error probability of all error candidate words in the source code and uses it to classify the source code as either correct or incorrect. The average precision, recall, and F-measure values of the proposed model are 0.97, 0.96, and 0.96, respectively, for the classification of source code with syntax errors and 0.91, 0.95, and 0.93, respectively, for the classification of source code with logic errors; these values are better than those for existing state-of-the-art models. The proposed model shows better performance for long sequences of source code compared to that for LSTM and RNN. Our model contributes to source code error assessment, detection, and classification, especially logic error detection and classification, for which conventional compiler fail. Furthermore, our model predicts the correct words in place of the error in the source code, making these predicted words helpful for students and programmers to quickly fix the incorrect code. In particular, newborn programmers will benefit more from the proposed model in learning programming. The proposed model has some limitations. Error assessment and detection accuracy are sometimes below the expected values. When the estimated error probability of an error candidate word is below 0.9, the proposed model does not consider this word as an error candidate even though it

might be an error. The experimental results obtained from the source code based on the C programming language do not ensure that the model's performance will be the same as using other programming languages. In the future, we will work to resolve these issues using bidirectional LSTM and other deep neural networks. The proposed model can be integrated with an online-based judge system to evaluate source code.

Author Contributions: Conceptualization, M.M.R., Y.W. and K.N.; Data curation, M.M.R; Formal analysis, M.M.R; Funding acquisition, Y.W.; Methodology, M.M.R and Y.W.; Resources, M.M.R; Software, M.M.R; Supervision, Y.W.; Validation, M.M.R; Visualization, M.M.R; Writing – original draft, M.M.R; Writing – review & editing, M.M.R, Y.W. and K.N. 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.

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

Data Availability: 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/ and http://developers.u-aizu.ac.jp/index.

References

1. Minku, L.L.; Mendes, E.; Turhan, B. Data mining for software engineering and humans in the loop. *Prog. Artif. Intell.* **2016**, *5*, 307–314. [CrossRef]
2. Monperrus, M. Automatic software repair: A bibliography. *ACM Comput. Surv. (Csur)* **2018**, *51*, 1–24. [CrossRef]
3. Seo, H.; Sadowski, C.; Elbaum, S.; Aftandilian, E.; Bowdidge, R. Programmers' build errors: A case study (at google). In Proceedings of the 36th International Conference on Software Engineering (ICSE '14), Hyderabad, India, 31 May–7 June 2014; pp. 724–734.
4. Li, Z.; Jing, X.-Y.; Zhu, X. Progress on approaches to software defect prediction. *IET Softw.* **2018**, *12*, 161–175. [CrossRef]
5. Ozakıncı, R.; Tarhan, A. Early software defect prediction: A systematic map and review. *J. Syst. Softw.* **2018**, *144*, 216–239. [CrossRef]
6. Catal, C.; Diri, B. A systematic review of software fault prediction studies. *Expert Syst. Appl.* **2009**, *36*, 7346–7354. [CrossRef]
7. Halstead, M.H. *Elements of Software Science (Operating and Programming Systems Series)*, 2nd ed.; Elsevier: Amsterdam, The Netherlands, 1977.
8. McCabe, T.J. A complexity measure. *IEEE Trans. Softw. Eng.* **1976**, *SE-2*, 308–320. [CrossRef]
9. Jureczko, M.; Spinellis, D.D. Using object-oriented design metrics to predict software defects. In *Models and Methods of System Dependability*; Oficyna Wydawnicza Politechniki Wrocławskiej: Wrocław, Poland, 2010; pp. 69–81.
10. Pu, Y.; Narasimhan, K.; Solar-Lezama, A.; Barzilay, R. Sk_p: A neural program corrector for mooc. In Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, Amsterdam, The Netherlands, 30 October–4 November 2016; pp. 39–40.
11. Pennington, J.; Socher, R.; Manning, C.D. Glove: Global vectors for word representation. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar, 25–29 October 2014; pp. 1532–1543.
12. White, M.; Vendome, C.; Linares-Vásquez, M.; Poshyvanyk, D. Toward deep learning software repositories. In Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15), Florence, Italy, 16–17 May 2015; pp. 334–345.
13. Bengio, Y.; Boulanger-Lewandowski, N.; Pascanu, R. Advances in optimizing recurrent networks. In Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, Vancouver, BC, Canada, 26–31 May 2013; pp. 8624–8628.
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. Nagappan, N.; Ball, T. Using software dependencies and churn metrics to predict field failures: An empirical case study. In Proceedings of the First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), Madrid, Spain, 20–21 September 2007; pp. 364–373.
16. Moser, R.; Pedrycz, W.; Succi, G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In Proceedings of the 30th International Conference on Software Engineering, Leipzig, Germany, 10–18 May 2008; pp. 181–190.
17. Arar, O.F.; Ayan, K. A feature dependent naive Bayes approach and its application to the software defect prediction problem. *Appl. Soft Comput.* **2017**, *59*, 197–209. [CrossRef]
18. Jing, X.-Y.; Ying, S.; Zhang, Z.-W.; Wu, S.-S.; Liu, J. Dictionary learning based software defect prediction. In Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May–7 June 2014; pp. 414–423.
19. Rahman, M.M.; Watanobe, Y. An efficient approach for selecting initial centroid and outlier detection of data clustering. In *Advancing Technology Industrialization Through Intelligent Software Methodologies, Tools and Techniques*; IOS Press: Amsterdam, The Netherlands, 2019; Volume 318, pp. 616–628.
20. Intisar, C.M.; Watanobe, Y. Classification of Online Judge Programmers based on Rule Extraction from Self Organizing Feature. In Proceedings of the 9th International Conference on Awareness Science and Technology (iCAST), Fukuoka, Japan, 19–21 September 2018; pp. 313–318.
21. Intisar, C.M.; Watanobe, Y. Cluster Analysis to Estimate the Difficulty of Programming Problems. In Proceedings of the 3rd International Conference on Applications in Information Technology (ICAIT '18), Aizu-Wakamatsu, Japan, 1–3 November 2018; pp. 23–28.
22. Alreshedy, K.; Dharmaretnam, D.; Germán, D.M.; Srinivasan, V.; Gulliver, T.A. SCC: Automatic Classification of Code Snippets. In Proceedings of the 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM), Madrid, Spain, 23–24 September 2018; pp. 203–208.
23. Ram, A.; Nagappan, M. Supervised Sentiment Classification with CNNs for Diverse SE Datasets. *arXiv* **2018**, arXiv:1812.09653.
24. Reyes, J.; Ramírez, D.; Paciello, J. 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), Las Vegas, NV, USA, 15–17 December 2016; pp. 514–519.
25. Terada, K.; Watanobe, Y. Automatic Generation of Fill-in-the-Blank Programming Problems. In Proceedings of the 2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), Singapore, 1–4 October 2019; pp. 187–193.
26. Tai, K.S.; Socher, R.; Manning, C.D. 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, Beijing, China, 26–31 July 2015; pp. 1556–1566.
27. Pedroni, M.; Meyer, B. Compiler error messages: What can help novices? In Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education, Portland, OR, USA, 12–15 March 2008; pp. 168–172.
28. Saito, T.; Watanobe, Y. Learning Path Recommendation System for Programming Education based on Neural Networks. *Int. J. Distance Educ. Technol. (Ijdet)* **2019**, *18*, 36–64. [CrossRef]
29. 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.
30. Fan, G.; Diao, X.; Yu, H.; Yang, K.; Chen, L. Software Defect Prediction via Attention-Based Recurrent Neural Network. *Sci. Program.* **2019**, *2019*, 6230953. [CrossRef]
31. Ohashi, H.; Watanobe, Y. Convolutional Neural Network for Classification of Source Codes. In Proceedings of the 2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), Singapore, Singapore, 1–4 October 2019; pp. 194–200.
32. Graves, A. Generating Sequences with Recurrent Neural Networks. *arXiv* **2014**, arXiv:1308.0850.
33. Mnih, V.; Heess, N.; Graves, A.; Kavukcuoglu, K. Recurrent models of visual attention. In Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS), Montreal, QC, Canada, 8–13 December 2014; pp. 2204–2212.

34. Luong, T.; Pham, H.; Manning, C.D. Effective approaches to attention-based neural machine translation. In Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP), Lisbon, Portugal, 17–21 September 2015; pp. 1412–1421.
35. Chen, J.; Zhang, H.; He, X.; Nie, L.; Liu, W.; Chua, T.-S. 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), Shinjuku, Tokyo, Japan, 7–11 August 2017; pp. 335–344.
36. Cheng, J.; Dong, L.; Lapata, M. Long short-term memory-networks for machine reading. In Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP), Austin, TX, USA, 1–5 November 2016; pp. 551–561.
37. Bahdanau, D.; Cho, K.; Bengio, Y. Neural machine translation by jointly learning to align and translate. In Proceedings of the 3rd International Conference on Learning Representations (ICLR), San Diego, CA, USA, 7–9 May 2015; pp. 1–15.
38. Li, J.; Wang, Y.; Lyu, M.R.; King, I. Code completion with neural attention and pointer networks. In Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI'18), Stockholm, Sweden, 13–19 July 2018; pp. 4159–4165.
39. Watanobe, Y. Aizu Online Judge. 2017. Available online: https://onlinejudge.u-aizu.ac.jp/ (accessed on 10 October 2019).
40. Aizu Online Judge. Developers Site (api). 2004. Available online: http://developers.u-aizu.ac.jp/index (accessed on 10 October 2019).
41. 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.
42. Kingma, D.P.; Ba, J. Adam: A Method for Stochastic Optimization. In Proceedings of the 3rd International Conference for Learning Representations (ICLR), San Diego, CA, USA, 7–9 May 2015; pp. 1–13.
43. Hinton, G. Deep belief networks. *Scholarpedia* **2009**, *4*, 2009. [CrossRef]