IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received February 1, 2022, accepted February 16, 2022, date of publication March 1, 2022, date of current version March 14, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3155558

Sequence-to-Sequence Learning-Based Conversion of Pseudo-Code to Source Code Using Neural Translation Approach

UZZAL KUMAR ACHARJEE[1], MINHAZUL AREFIN[1], (Member, IEEE), KAZI MOJAMMEL HOSSEN[1], (Member, IEEE), MOHAMMED NASIR UDDIN[1], MD. ASHRAF UDDIN[2], AND LINTA ISLAM[1]
[1]Department of Computer Science & Engineering, Jagannath University, Dhaka 1100, Bangladesh
[2]Internet Commerce Security Laboratory, Centre for Informatics and Applied Optimisation, Federation University, Ballarat, VIC 3350, Australia

Corresponding author: Minhazul Arefin (minhazularefin21@gmail.com)

This work was supported by the Jagannath University, Dhaka, Bangladesh.

ABSTRACT Pseudo-code refers to an informal means of representing algorithms that do not require the exact syntax of a computer programming language. Pseudo-code helps developers and researchers represent their algorithms using human-readable language. Generally, researchers can convert the pseudo-code into computer source code using different conversion techniques. The efficiency of such conversion methods is measured based on the converted algorithm's correctness. Researchers have already explored diverse technologies to devise conversion methods with higher accuracy. This paper proposes a novel pseudo-code conversion learning method that includes natural language processing-based text preprocessing and a sequence-to-sequence deep learning-based model trained with the SPoC dataset. We conducted an extensive experiment on our designed algorithm using descriptive bilingual understudy scoring and compared our results with state-of-the-art techniques. Result analysis shows that our approach is more accurate and efficient than other existing conversion methods in terms of several performances metrics. Furthermore, the proposed method outperforms the existing approaches because our method utilizes two Long-Short-Term-Memory networks that might increase the accuracy.

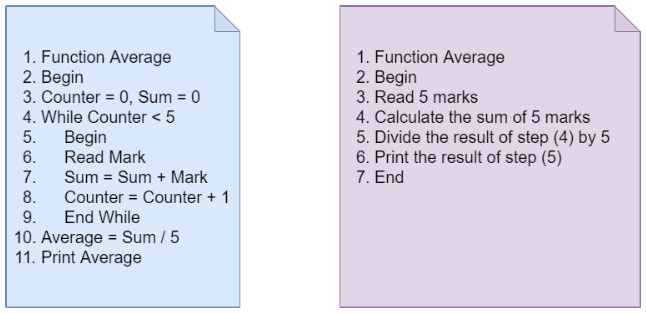
INDEX TERMS Sequence-to-sequence learning model, natural language processing, pseudo-code, machine translation, source code.

I. INTRODUCTION

In modern times, researchers and academicians tend to represent problem-solving procedures in the form of algorithms, more specifically pseudo-code [1]. Pseudo-code is a language that enables programmers to design algorithms. Usually, pseudo-code might be coded using natural language details, which allows a beginner-level programmer to figure out the algorithm appropriately. On the other hand, a novice programmer might find it hard to manually translate pseudo-code to source code [2]. Further, such translations can also save time for a researcher by converting the provided pseudo-code to source code to verify the correctness of an algorithm. To facilitate this translation, researchers have already proposed different methods with various levels of accuracy. This paper aims to provide a framework for converting pseudo-code statements into appropriate source code [3], [4]. The proposed approach uses a deep learning method to translate pseudo-code statements into source code [5]. The suggested method is based on a sequence-to-sequence (Seq2Seq) learning model [6] that can realize pseudo-code and intermediate code translation. However, most state-of-the-art works transform pseudo-code statements to executable code using natural language processing techniques that are unable to learn new statements over time [7], [8]. Many research works have used the SpoC dataset that contains 26% blank pseudo-code. We address this problem by replacing blank space with some specific string before feeding this dataset into our model. The syntax and structure of pseudo-code used to present algorithms in information technology varies. For instance, the terms ''output'' and ''display'' may represent the meanings of ''write'' and ''view,'' respectively.

The majority of recent pseudo-code is written in the style of natural languages. Further, the ambiguity of pseudo-codes

The associate editor coordinating the review of this manuscript and approving it for publication was Jerry Chun-Wei Lin.



```
1. Function Average
2. Begin
3. Counter = 0, Sum = 0
4. While Counter < 5
5.     Begin
6.     Read Mark
7.     Sum = Sum + Mark
8.     Counter = Counter + 1
9.     End While
10. Average = Sum / 5
11. Print Average
```

```
1. Function Average
2. Begin
3. Read 5 marks
4. Calculate the sum of 5 marks
5. Divide the result of step (4) by 5
6. Print the result of step (5)
7. End
```

FIGURE 1. Two different pseudo-codes presentation for a specific algorithm.

might exist in tokens, assertions, and expressions. The translation of pseudo-codes into source code is difficult due to the lack of complete standards and rules [10]. Figure 1 shows two different pseudo-codes presentations for a particular statistical algorithm. To address this ambiguity, there needs to design an effective algorithm to translate pseudo-code to source code, and we propose a neural translator based on the sequence to sequence learning method.

To map the terms of pseudo-code with the syntax of a particular programming language semantics is challenging due to their unbounded properties [11]. A look-up table for mapping tokens between pseudo-code and source code might not help in the translation process because there exists ambiguity in the tokens. To solve this problem, we adopt Natural Language Processing (NLP) techniques with a deep learning approach. Adopting natural language processing can minimize the shortcomings in fundamental automated software development for non-expert users [12]. There needs a sophisticated translation tool to handle the diverse structure of pseudo-code. Our study attempts to automate the compilation of the pseudo-code programming language with higher efficiency.

Our contributions are summarized as follows.

- Designing an algorithm for converting pseudo-code into source code using natural language processing and sequence to sequence deep learning module.
- Implementing the proposed translation algorithm and comparing the performance of our approach with the state-of-the-art works. Our findings show that a neural translation-based deep learning approach can outperform other existing systems that leverage rule-based methods and advanced programming knowledge.
- Solving the SPoC dataset's 26% blank pseudo-code problem by replacing blank space with some specific string before feeding this dataset into our model.

The rest of the paper is organized as follows. In Section II, we review state-of-the-art works in pseudo-code translation. The proposed framework for converting pseudo-code to source code is presented in Section III. The performance analysis of the proposed approach is done in Section IV before concluding the paper in Section V.

II. LITERATURE REVIEW

Many researchers have already devised diverse kinds of techniques to convert pseudo code statements into source code. Recently, Kulal *et al.* [13] proposed a standard LSTM encoder and decoder to translate pseudocode to a computer programming language. First, each pseudocode line is translated as a separate part of the program. Next, all the potential combinations of the candidates are examined until the source code successfully passes all the error functions. The attentive-based coping mechanism and the coverage vector were also employed in this model. After the pseudocode is translated, beam searches are applied to create a sorted list of possible translations in which each string is a string token sequence. However, this model has several limitations. If there is an ambiguity in any pseudocode, the code generation will be ambiguous as well [14]. Moreover, there is a significant problem in declaring the variable types. If the method cannot detect the variable type correctly, the pseudocode will be completely wrong. To solve these issues, Bi-directional Long-Short-Term-Memory (LSTM) encoder and decoder can be used to generate the program. They also suggested allocating credit based on signals from compilation mistakes, which account for 88.7% of all code failures.

Zhong *et al.* [15] proposed a translation tool based on semantic scaffolds by adding semantic constraints to translate the pseudocode into a programming language or source code. This method's hierarchical beam search algorithm can integrate with all the constraints by performing efficiently and giving better search space coverage than the standard approaches. By applying this search method to the SPoC dataset, this method can generate the code from the pseudocode where only natural language pseudocodes are given, achieving a state-of-the-art 55.1% accuracy. Though beam search can produce the top solution, the time complexity of this search algorithm increases exponentially with the increase of pseudocode's length. In the worst-case scenario, the beam search algorithm is often skewed toward variations at the finishing end of the program because of its greedy nature. This skewness causes the wastage of its resources on the wrong candidates. They obtained a 10% absolute improvement in top-100 accuracy over the prior state-of-the-art by leveraging semantic scaffolds during inference.

Imam and Alnsour [16] proposed an automated translation system emphasizing Natural Language Processing (NLP) to translate the programming language from the pseudocode. This hybrid approach (Code-Composer) [17] can compose the language automatically by transforming pseudocode into the Cprogramming language. The CodeComposer uses various NLP techniques to examine the pseudocodes, including verb categorization, semantic role-marking, and thematic roles. Moreover, the coding composer is an intelligent computer assistance software engineering tool that converts the pseudocode into a C# declaration in the.Net environment using a semantic rules-based mapper [18].

By applying the binomial techniques, this system provides a precision of 88% though it is only applicable to a distinct language. Moreover, the system might produce mistranslated (MsT) and non-translated (NoT) errors. CodeComposer has a precision of 88%, a recall of 91%, and an F-measure of 89%, according to a binomial approach evaluation of its accuracy. Thus, the CodeComposer needs human revision to complete the translation process. By using semantic role labeling [19] with more semantic roles, the issues mentioned above could be solved.

Teduh Dirgahayu *et al.* [20] presented a conceptual-metamodeling approach that uses model-driven engineering (MDE) [21] to develop an automatic translation tool for translating pseudocode into source code. By introducing an intermediate model, this translation tool can disjoin the pseudocode statements in XML. The utilization of previously produced translation tools can assist us in constructing more efficient translation tools. This method could only produce the source code in C++ from the pseudocode written in Bahasa Indonesia. However, applying this method cannot translate the functions and procedure calls. Further, this model cannot interpret the complex data structures, including arrays and lists.

NU Koyluoglu *et al.* [22] proposed a technique that hassles through using transformers for the mission of pseudocode-to-C++-code translation and did a comparative study with the earlier posted effects on the usage of LSTMs. They hire a couple of architectures [23], tokenizers, and employ pre-trained English language fashions to enhance training. They additionally discover the effects of various kinds and quantities of contexts on our fashions. Using useful correctness for overall performance assessment as hostile to conventional methods, our effects shape the overall performance of preceding paintings closely and factor in the extra advantage of context in line-through-line translations. Guang Yang *et al.* [24] endorse a singular deep pseudo-code learning approach (Deep-Pseudo) through code function extraction and Transformer. In particular, Deep Pseudo [25] makes use of a Transformer encoder to carry out encoding for supply code, after which it uses a code function extractor to research the expertise of nearby features. Finally, it uses a pseudo-code generator to carry out decoding, which could generate the corresponding pseudo-code. It picks out corpora (i.e., Django and SPoC) from real-international large-scale tasks as our empirical subjects [26]. They first evaluate Deep Pseudo with seven ultramodern baselines from the pseudo-code era and neural device translation domain names in terms of four overall performance measures. The results display the competitiveness of Deep Pseudo. Moreover, they additionally examine the rationality of the aspect settings in Deep Pseudo.

III. PROPOSED METHODOLOGY

This section describes every step of our proposed pseudo-code conversion. The proposed framework to convert pseudo-code to source code is presented in Figure 2.

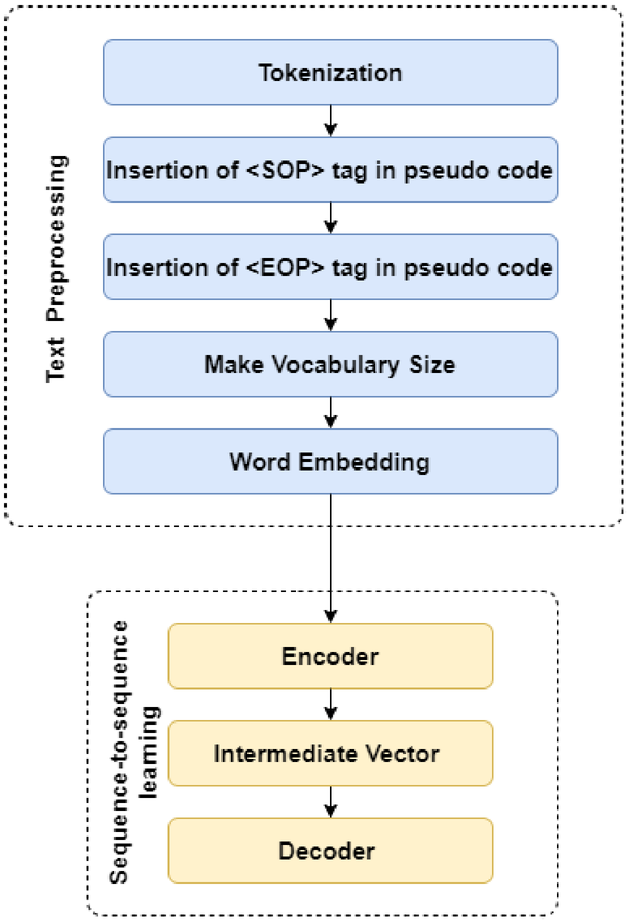

Text Preprocessing
Tokenization
Insertion of <SOP> tag in pseudo code
Insertion of <EOP> tag in pseudo code
Make Vocabulary Size
Word Embedding
Sequence-to-sequence learning
Encoder
Intermediate Vector
Decoder

FIGURE 2. Proposed pseudo-code to source code conversion system.

Our approach includes input preprocessing, reshaping the data, splitting the dataset, and sequence-to-sequence learning. The algorithm of our pseudo-code to source code translation system is presented in Algorithm 1. In addition, the description of the algorithm is provided in Figure 2.

A. TEXT PREPROCESSING

Text preprocessing is a natural language processing (NLP) technique that transforms unprocessed text into a recognizable format. However, source or unprocessed data might have inadequacy, irregularity, and many other flaws. This step has addressed these problems. Our text preprocessing pipeline includes tokenization, inserting <SOP> tag, and <EOP tags, defining vocabulary size, and embedding words.

1) TOKENIZATION

Tokenization splits the string of pseudo-code statements into tokens or unique words. We use the Sentence Boundary Disambiguation (SBD) method to extract a list of tokens from the pseudo-code. In the deep learning approach, preprocessing has been performed on a pre-trained stage using language-specific algorithms such as the Text to Word Sequence Models from Keras that run on top of the Tensor Flow python package.

Algorithm 1: Algorithm of the Proposed Pseudo-Code to Source Code Conversion

```
Input: P = Pseudo-code; t = Time-step
Output: s = Source code
Split P into lines
for $i = 0$ to line in P do
    Split Line into Tokens
    P[i] = "SOP" + Tokens + "EOP"
    $V_s$[i] = P[i].length
    let f be a list of tuples(values, frequency)
    for $j = 0$ to max-length(P) do
        freq ← 0
        for $k = 0$ to line in P do
            $f_o$[k] ← $FrequencyOfOccurance(j, k)$
            freq[k] ← $freq[k]$ + $f_o$[k]
        end
        f[j] ← $append(j, freq[j])$
    end
    f[i] ← sort f based on frequencies
    w[i] ← $Word2Vec(f[i], V_s[i])$
    x[i] ← $InputShape(P[i], t, w[i])$
end
for $i = 0$ to line in P do
    e[i] ← $Encoder(x[i])$
    s[i] ← $Decoder(e[i])$
end
```

2) INSERTION OF <SOP> TAG IN PSEUDO CODE

At the beginning of the pseudo-code statement, we add a starting tag called $< SOP >$. This tag facilitates tokenizing the pseudo-code, which is a substantial step for the decoder of the Seq2Seq process. The Seq2Seq approach initiates its embedding step after tokenizing.

3) INSERTION OF <EOP< TAG IN PSEUDO CODE

The encoder of Seq2Seq reads the pseudo-code and makes a mathematical representation of the pseudo-code sequence. A sequence is ended with an "EOP" tag. The EOP token is as important as the SOP tag. The EOP token diffuses irrational-length pseudo-code. Without the EOP tag, we cannot know when the decoder step ends, and garbage output is produced without the EOP tag.

4) DEFINING VOCABULARY SIZE

The vocabulary content of a dataset is commonly determined by the statistics of the unique word uni-grams. Once the algorithm of the pseudo-code conversion is executed, the text has been processed and relevant metadata is required to collect and store. We considered the diameter of the pseudo-code string and the sum of the unique token for making vocabulary. The vocabulary words are like a dictionary. It will ensure the top frequencies tokens being saved in an array. By that, it reduces the time for converting tokens into vectors. Its main importance is to save vectors of different types of programming language's keywords.

5) WORD EMBEDDING

Word embedding is the mathematical representations of words as vectors. They are created by analyzing a body of text and representing each word, phrase, or entire document as a vector in a high-dimensional space. Each token or word in the skip-gram model is provided in two $\delta - dimension$ vectors. These vectors are used to determine the possibility of dependent tokens. We assume that each token is indexed as i in the dictionary, where the token vector is indicated as $v_i \in R^d$, and $u_i \in R^d$. w_c and w_o are the token of the framework as mutual token c and o in the dictionary.

The conditional probability of generating the context word for the given central target word can be obtained by performing a softmax operation on the vector inner product according to equation 1.

$$P(w_o|w_c) = \frac{exp(u_o^T v_c)}{\sum_{i \in v} exp(u_i^T v_c)} \tag{1}$$

Here, vocabulary or tokens of pseudo-code index is set as $V = 0, 1, \ldots, |V| - 1$.

We assume that T is a length of pseudo-code sequence, wherein t represents time required to make token which is denoted as $w(t)$. If all tokens in the framework are generated following the central tokens and framework's token size is m, the skip-gram model is described as a function with the shared tendency of creating all framework tokens divided any central token.

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m, j \neq 0} P(w^{(t+j)}|w^{(t)}) \tag{2}$$

Here, all time-step that is between 1 to T can be disregard. The skip-gram approach [27] have two parameters. One is central goal token and another is for framework token for each entity. In the pre-process stage, the parameters are taught by expanding likelihood estimation or function. If the loss function is minimized, the following equation 3 is obtained.

$$-\sum_{t=1}^{T} \sum_{-m \leq j \leq m, j \neq 0} log(P(w^{(t+j)}|w^{(t)})) \tag{3}$$

We require to take the shorter pseudo-code sequence using random sampling at each time step if SDG is applied. This process is performed to find the loss of the sub-sequence of pseudo-code. First, we compute the gradient value in the proposed algorithm. Next, word embedding model are updated. We calculate the slope of the logarithmic provisional possibility principal token vector and the fundamental token vector which is the main function for calculating gradient. The equation 4, 5, 6 and 7 show the gradient calculation.

$$log(P(w_o|w_c)) = u_o^T v_c - log(\sum_{i \in v} exp(u_i^T v_c)) \tag{4}$$

By differentiating the formula 4, the gradient v_c is derived as equation 5:

$$\frac{\delta log(P(w_o|w_c))}{\delta v_c} = u_o - \sum_{j \in v} \frac{\sum_{j \in v} exp(u_j^T v_c) u_j}{\sum_{i \in v} exp(u_i^T v_c)} \tag{5}$$

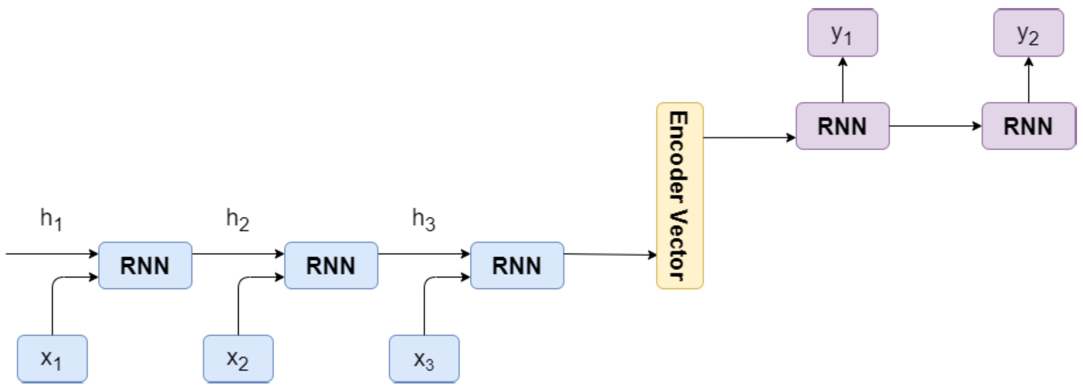


FIGURE 3. Structure of sequence-to-sequence learning.

$$\frac{\delta log(P(w_o|w_c))}{\delta v_c} = u_o - \sum_{j\in v}(\frac{exp(u_j^T v_c)}{\sum_{i\in v} exp(u_i^T v_c)})u_j \quad (6)$$

$$\frac{\delta log(P(w_o|w_c))}{\delta v_c} = u_o - \sum_{j\in v} P(w_j|w_c)u_j \quad (7)$$

The proposed system finds the central goal token w_c from pseudo-code with the dictionary tokens or words. We use the identical function to find the gradients. The identical function finds gradients from other tokens vectors. If we assume a token as i after training, we get two identical token vector sets: one is v_i and the other one is u_i. In NLP, the central goal token represents the mathematical representation of a token or word. A token vector serves as a token or word. Here word embedding converts the tokens into vectors. Word2Vec model has two sub-model as the continuous bag of words (CBOW) and skip-gram models.

The skip-gram model predicts the next token based on the central goal token from the pseudo-code. On the other hand, CBOW uses the next token to find the central goal token. Concepts are easier to grasp using Word2vec. Using word2vec [27] is simple and has a solid design. Compared to other methods, this method can be rapidly trained because of not requiring user's manual input. This approach works well for both a small number of datasets and a big number of datasets. Word2vec captures semantic similarity very well. This method works faster because of being uncontrolled without users' efforts.

B. SEQUENCE-TO-SEQUENCE LEARNING

Sequence-to-sequence learning (Seq2Seq) is a state-of-the-art deep learning method that trains models [28] to convert sequences from one domain (e.g., a pseudo-code statement) to another domain (e.g., a source code of a programming language). Today, Google Translate, speech devices and chatbot-like internet apps have the potential for this approach. The Seq2Seq approach can handle sequence-based problems, especially when inputs and outputs differ. In the general case, input pseudo-code and output source code have different lengths. The entire input sequence is required to start predicting the target. The structure of sequence-to-sequence learning is presented in Figure 3.

Preceding Seq2Seq [29], phrase-based systems were mostly used. Inputs and outputs in sequences of sentences can be considered for a sentence-based translation system. However, long-term dependence is still hard to represent. Seq2Seq, particularly by using LSTM, has the benefit of arbitrarily creating sequences after seeing the whole entry, with the usage of contemporary translation systems. In order to provide a practical translation, they can even automatically focus on certain sections of the input.

An encoder that takes the input sequence of the model as an input and codes it into a "context vector" of a defined size. A decoder that utilizes the above context vector as a seed to create an output sequence. This is why Seq2Seq models are typically called encoder decoder models. These two networks will be examined individually for the specifics.

1) ENCODER

An encoder is known as the "stack thereof," based on a Recurrent Neural Network (RNN). Encoder takes pseudo code statement as input and returns their intramural shape. The RNN, redeeming the original state provides the factors of the decoder in the next step. In our case, the input is a pseudo-code statement sequence, and the corresponding output is a programming language source code. Each unit acts as an LSTM cell to decrease gradient calculation. Which makes it perform better on long sequences. As described above, every text or string is in the form of indices, where every token in the pseudo-code line is depicted by a unique index number given to it in the pre-processing phase. A sequence of hidden recurrent neural networks is made by LSTM or GRU cells. Each cell takes a single token from input pseudo-code statement, collects necessary information from the input, and propagates it to the forward step.

In this translation system, the input pseudo-code statement is full of tokens. Here, each token is defined as X_i where i is defined as the token sequence. The hidden LSTM or GRU cells h_i are calculated as equation 8:

$$h_t = \int (W^{(hh)}h_{(t-1)} + W^{(hx)}x_t) \quad (8)$$

This equation represents the general RNN. In this equation, with the previously utilized hidden cell, we only add suitable

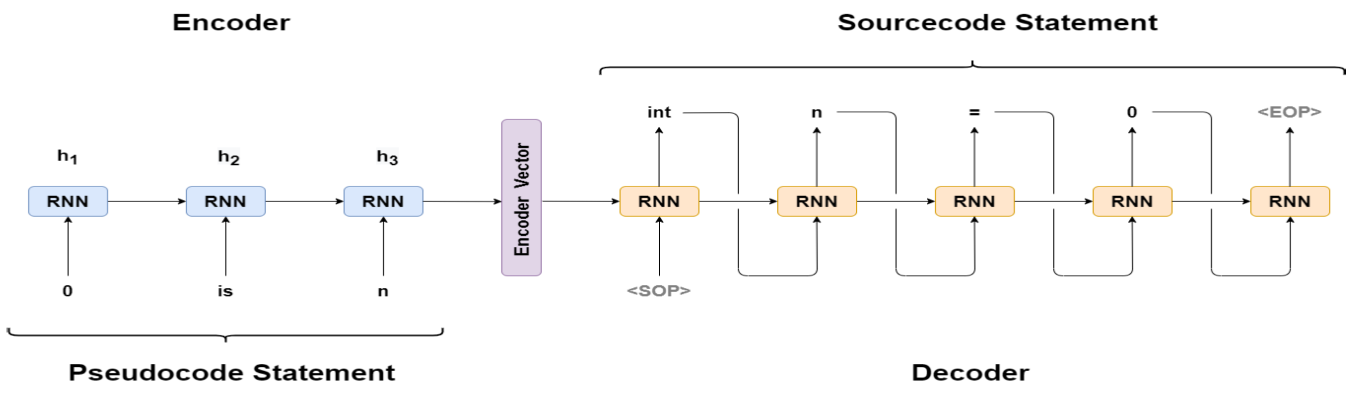


FIGURE 4. Process of translating pseudo code into source code.

weights marked as $h_{(t-1)}$ and the vector input differentiated as a x_t.

2) INTERMEDIATE (Encoder) VECTOR

Attention Mechanism or Intermediate (Encoder) Vector is the last step in the hidden LSTM cell. It uses the output of the previous step. The previous output is calculated according to equation 8. This step of the encoder gathers all the information from the pseudo-code statement to aid the decoder step in making accurate source code. It works as the first hidden cell for the next part of the model. We use the global dot product attention to compute the context vector C_i.

$$h_t = \int (W^{(hh)} h_{(t-1)} + W^{(hx)} x_t) \tag{9}$$

3) DECODER

The decoder is also called the "stack thereof," based on another Recurrent Neural Network (RNN). The RNN is qualified to forecast the source code's next character based on the source code's previous characters. Specifically, this neural network is qualified to convert pseudo code into source code into a similar sequence step by step. This type of training process is termed "teacher forcing." Generally, the encoder step uses state vectors that are gained from the pseudo-code in the first state. However, the decoder gets all of the necessary data about the source code. Decoder with more than one recurring neural cell in each case predicts a y_t output at t at any moment. Each cell accepts the preceding cell's secret status and forecast its hidden cell output.

In converting a problem, the output source code collects keywords, variables, and mathematical expressions. Each token is noted as y_i, where i is the token sequence. Any hidden cell h_i is calculated using the equation:

$$h_t = \int (W^{(hh)} h_{t-1}) \tag{10}$$

In this case, we just calculate the following one using the previously concealed cell. Whenever t is computed, the decoder step output is y_{tat}:

$$y_t = softmax(W^s h_t) \tag{11}$$

TABLE 1. Description of dataset.

Dataset	SPoC
Types of natural language input	pseudo-code
Programming Language	C++
Granualarity of text description	line
Total numbers of program	18,356
Average lines per program	14.7
Fraction of human-annotated text	100%
Number of annotators (total)	59
Number of test cases (average)	38.6

IV. EXPERIMENTAL ANALYSIS

In addition to the SPoC dataset, our proposed model has used two types of data: (1) Pseudocode statement based on algorithms and (2) the appropriate source code based on a programming language. This dataset is used to test the translation abilities and calculate the learning ability from the Seq2Seq method of our proposed model. The description of the datasets is illustrated in Table 1.

Before tokenization of the phrases, we have to determine the length of the phrases. We will record the length of the all sentences in two lists for both pseudocode and source code. The figure 6 illustrates a blue histogram which shows that the source code length is 23 while the pseudo-code length is 26.

There are two primary distinctions between the single integer and text embedding. A word can be represented with a single integer or by a vector of any size. Thus, word embedding can capture more word information than a single integer. Moreover, a single integer can not capture the links between distinct words. Conversely, word embedding retains the connections between words.

For this reason, we will apply Word2Vec word embedding for the translated source code in the output. The complete sequence of steps of Seq2Seq to achieve source code is shown in figure 4. The goal of the translation process is to separate variables and keywords from the pseudocode statements. A pseudo-code statement is a compact way of describing an algorithm.

A. TEXT PRE-PROCESSING

Our proposed system starts with text pre-processing step. Tokenization is the first step of text pre-processing. Given a

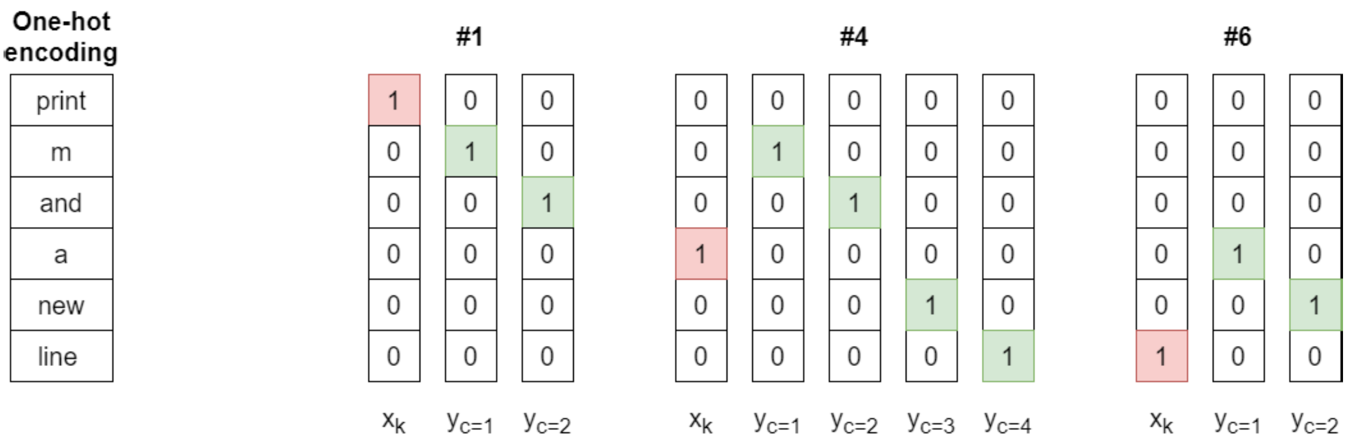


FIGURE 5. One-hot encoding for our sample dataset.

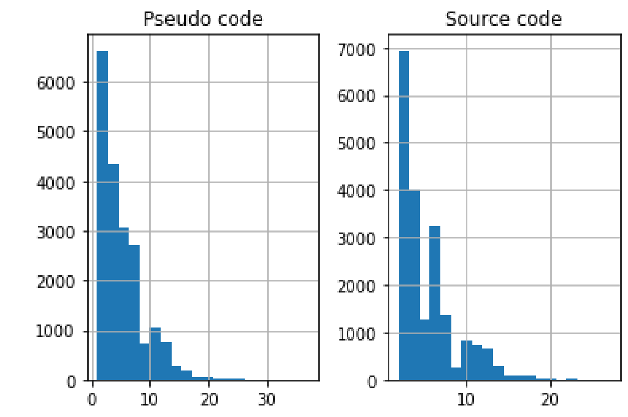


FIGURE 6. Maximum length of pseudo-code and source code.

pseudo-code statement, the task of tokenization is to break the statements into tokens and remove key characters like punctuations simultaneously. An example of tokenization is:

Input: n is 0

Output: [‘n’, ‘is’, ‘0’]

For seq2seq model (Pseudo-code to source code translation), we have the encoder input data, that is, pseudo-code statement without <start> and <stop> tags. We have to add a tag to easily detect the start and end of the line for the seq2seq model. For this reason, we added start of pseudo-code (SOP) tag and end of pseudo-code tag (EOP). Our data is organized in such a manner:

Decoder input data: Contains pseudo-code statement which have both <SOP> and <EOP> tags.

Decoder target data: Contains source code statement which do not have any tag.

For example:

Input: ‘n’, ‘is’, ‘0’

Output: [‘SOP’, ‘n’, ‘is’, ‘0’, ‘EOP’]

1) WORD EMBEDDING

To convert the text into vectors using Word2Vec, most NLP embedding techniques allow text processing using

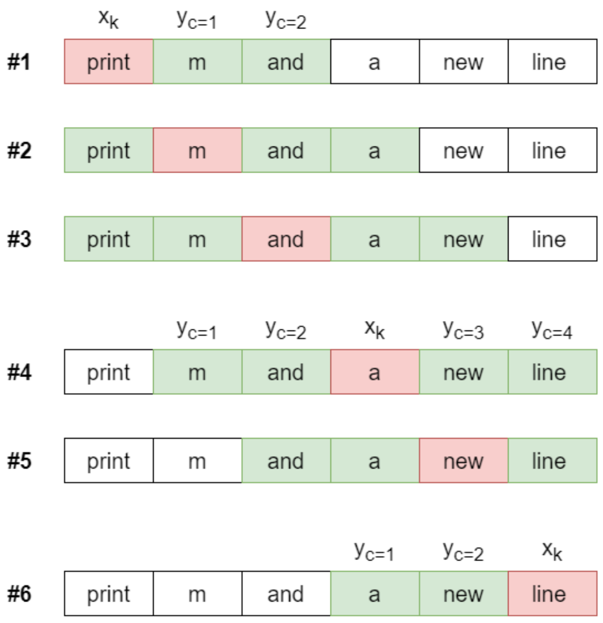


FIGURE 7. An example of sliding window.

linear algebra operations. Vectorizing text data enables us to construct prediction models using these vectors as inputs. We already grasp 90 % of the word2vec model if we comprehend both forward and backpropagation for Neural Networks.

The notion of a central word with context words may be likened to a sliding window that passes through the corpus of text. An example of this is “print m and a new line,” using a $C = 5$ window size (two before, and two after the center word). When the context window glides over the sentence, the matching words fill it. When the context window reaches the borders of the sentences, the furthest window locations are simply dropped. The following figure 7 shows how this procedure looks. Please note the center word is now x_k and the context words are y_c instead of $w(t)$, $w(t+1)$, etc.

We can not apply single-hot encoding as the text data can not be transmitted straight through a matrix. This indicates

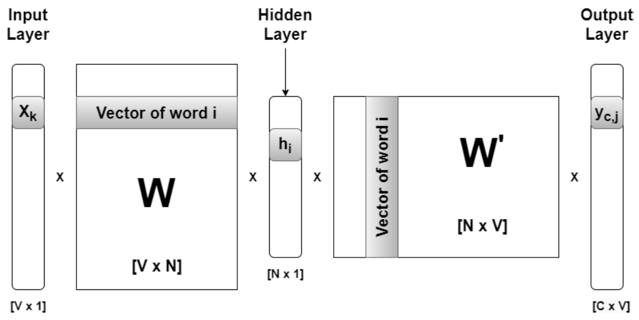


FIGURE 8. Skip-gram model architecture.

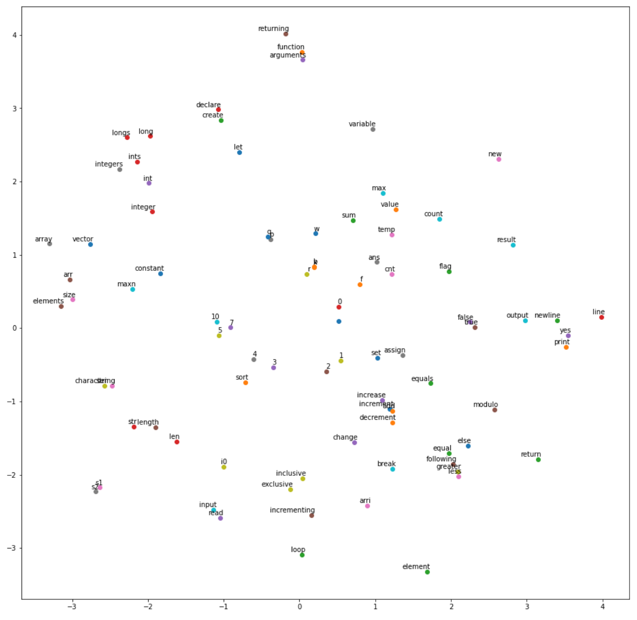


FIGURE 9. Selective model for Word2Vec representation.

that the vector v is the total number of single words in the corpus text (or shorter). For every word in this vector, v_n is zero everywhere, and except than v_n that is one. For every word that is in this vector, v_n is zero everywhere inside the v vector, except than v_n that is one. Every word is a one-size-fits-all location. The single-hot encoding of example 1, 5, and 9 above is provided in Figure 5

For future reference, the column vectors $y_{c=1}, \ldots, y_{c=C}$ are referred as panels. In the next step we have to load and train this data into the network. Most descriptive literature used the same graph to describe skip-gram, which shows in Figure 8 the last layer of the model with three or more matrices.

The classification portion (i.e. extraction of token probabilities from a vector representation of a text) is similar to neural classifiers and language models. Vector representation of text has a dimension of d, however in the end, a vector of size $|V|$ is required (probabilities for $|V|$ tokens/class). A linear layer can be used to transform a d-sized vector into a $|V|$-sized one.

A selective model for Word2Vec representation is described in Figure 9. We present the tokens of dataset with the minimum count of 1000. These examples indicate that each center word's output probabilities are divided pretty evenly across the right context word.

TABLE 2. A sample of training dataset.

Line No.	Pseudo-code	Source Code
1	function start	int main() {
2	s and t = strings	string s, t;
3	read s and t	cin >> s >> t;
4	set integer d1 to s[0] - t[0]	int d1 = s[0] - t[0];
5	assign value s[1] - t[1] to the integer d2	int d2 = s[1] - t[1];
6	create integer m with value max of abs(d1) and abs(d2)	int m = max(abs(d1), abs(d2));
7	c1 = 'L' if d1 > 0 or 'R' otherwise	char c1 = d1 > 0 ? 'L' : 'R';
8	c2 = 'D' if d2 > 0 or 'U' otherwise	char c2 = d2 > 0 ? 'D' : 'U';
9	print m and a new line	cout << m << endl;
10	for integer i = 0 to min of abs(d1) and abs(d2) exclusive,	for (int i = 0; i < min(abs(d1), abs(d2)); i++)
11	print c1, c2 and a new line	{ cout << c1 << c2 << endl; }
12	set d1 to abs(d1)	d1 = abs(d1);
13	set d2 to abs(d2)	d2 = abs(d2);
14	if d1 is less than d2	if (d1 < d2) c1 = c2;
15	for i = 0 to abs(d1 - d2)	for (int i = 0; i < abs(d1 - d2); i++)
16	exclusive print c1 and a new line	{ cout << c1 << endl; }
17		return 0; }

B. RESHAPE THE DATA TO NEURAL NETWORK SHAPE

Data restructuring is a frequent and laborious effort to manipulate and analyze real-life data. A dataset might come with various group levels and we need to redirect some sorts of analysis. Data sets may be large or lengthy in layout. Multiple rows indicate the record of one topic in a lengthy design, but in a broad layout one row is a record of one subject. To satisfy the criteria of statistical analysis, we need to be able to rearrange data smoothly and fluidly. Data reshaping is the reorganization of the data form without altering the dataset content.

There is an argument called 'input shape'' in the first hidden state of the LSTM layer. This must be a three-dimensional layer. The first one is sampling. One pseudo-code line is one sample. So, a pseudo-code contains many samples. The second one is time steps in the LSTM layer. One step for every attention it needs. The third one is characteristics. So we can say that the matrix contains one characteristic of a pseudo-code line at a single time step.

C. SPLIT DATA FOR TRAINING AND TESTING

The training phase is the first step for the Seq2Seq learning approach. The Seq2Seq deep learning Network is trained with the SPoC dataset. The Seq2Seq method uses the supervised approach for translating pseudo-code into source code. The SPoC training sub-dataset has all the characteristics needed to translate pseudo-code into source code. LSTM networks is used. The layers of LSTM are connected by a bi-directional pattern with the nearest neurons as an outcome of the training.

Our technique exploits the ''train test split'' function of the scikit-learn library to divide the SPoC dataset into two sub-datasets. The proposed system is prepared with the training sub-dataset from the dataset. The training data set consists of portion (80%) of the data set. The remainder(20%) are regarded as test data. This data is imported as.txt file. Part of the training data set is represented in Table 2.

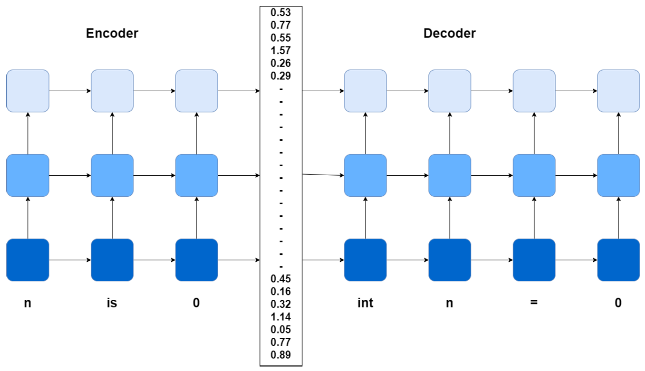


FIGURE 10. Seq2Seq network with encoder-decoder.

TABLE 3. An example of encoder and decoder's operations during training process.

Steps	Generated Code
1	main function $\longrightarrow Encoder \longrightarrow enc(h_1, c_1)$ enc$(h_1, c_1) + < sop > \longrightarrow Decoder \longrightarrow int + dec(h_1, c_1)$
2	enc$(h_1, c_1) + int \longrightarrow Decoder \longrightarrow main + dec(h_2, c_2)$
3	enc$(h_2, c_2) + main \longrightarrow Decoder \longrightarrow ()\{+dec(h_3, c_3)$
4	enc$(h_3, c_3) + ()\{ \longrightarrow Decoder \longrightarrow < eop > +dec(h_4, c_4)$

D. SEQUENCE-TO-SEQUENCE LEARNING

The remaining part of the vector uses the softmax operation to transform the raw numbers into token probabilities. In this proposed system, we have used two RNNs (LSTM) for encoder and decoder. Encoder RNN reads the source phrase. The end state is the decoder RNN's starting state. The goal is to translate all source information and to generate the target phrase using this vector. There are multiple levels for the encoder and the decoder. A multi-layer model like Sequence-to-Sequence-to-Network Learning is one of the earliest efforts to tackle sequence-to-sequence tasks with neural networks. The conditional language modelling is more than a technique to deal with sequence-to-sequence problems. Seq2Seq models are trained to predict the probability distribution of the next token given to the prior context, like neural LSTM models in Figure10. We need to optimize the probability of allocating the proper token at each stage of this model.

During the training phase, the true inputs to the decoder are all the output words of the sequence. An example of this training phase is shown in table 3. We assumed a penalty main function and the following phrase will be translated as follows:

The model is trained on the decoder input and output. The following word is anticipated by the preceding word as it was predicted in the previous time phase. The complete output sequence is unknown, while making actual prediction, thus, we have to predict the sequence. During prediction, we use <sop>, as all the output phrases start with <sop>. Table 4 is an illustration of how the prediction takes place. The line "main function" will be translated again:

Step 1 showed the hidden encoder status, cell status, and the <sop> as the decoder input. The decoder predicts a word y_1 that can be true or not. According to our model, the likelihood is 0.9413. In step 2, the cell and cell state concealed from step 1 together with y_1 are used as the decoder input for predicting y_2. This step is continued until the token <eop> is found. All predicted decoder outputs are then linked to the final output.

TABLE 4. An illustration of encoder and decoder's operations during testing.

Steps	Generated Code
1	main function $\longrightarrow Encoder \longrightarrow enc(h_1, c_1)$ enc$(h_1, c_1) + < sop > \longrightarrow Decoder \longrightarrow y_1(int) + dec(h_1, c_1)$
2	enc$(h_1, c_1) + y_1 \longrightarrow Decoder \longrightarrow y_2(main) + dec(h_2, c_2)$
3	enc$(h_2, c_2) + y_2 \longrightarrow Decoder \longrightarrow y_3((){}) + dec(h_3, c_3)$
4	enc$(h_3, c_3) + y_3 \longrightarrow Decoder \longrightarrow y_4(< eop >) + dec(h_4, c_4)$

TABLE 5. Examples of Seq2Seq based conversion process alongside the ground truth translations.

Pseudo-code	Original Source Code	Converted Source Code	Accuracy
function start	int main(){	int main()	95.24%
a and b = integer	int a, b;	int a, b;	100%
read a and b	cin >> a >> b;	cin >> a >> b;	100%
set integer c = a + b	int c = a + b;	int c = a + b;	100%
print c and a new line	cout << c << endl;	cout << c << endl;	100%
	return 0;}	return 0;	94.74%

We have selected a sequential to sequential learning approach as seq2seq is particularly well equipped to convert pseudo-code statements. In this section, we examine the core concept underlying the conversion process. We use the seq2seq learning method for the translation process. In the translation process, we handle individual operations by observing sample pseudo-code. Finally, we found a specific algorithm for the pseudo-code statement.

E. OUTPUT

The output for the analyzed test data is presented in Table 5. The test data contains the pseudo code as well as appropriate source code. Tags are used in this dataset for testing and converting each appropriate pseudo-code by the sequence to sequence learning method. The output contains each converted source code with original source code and inputted pseudo-code, along with the percentage of converted source code. However, some limitation still exists as the attention mechanism is not used in this model.

To apply this reasoning, we modify our model. We observe that the encoder's functionality is same as before. The sentence is processed through the encoder; and the hidden state in the original language and the cell state is the encoder's output.

We used some strings to remove the blank pseudo code problem. Here we mainly saw that the blank pseudo code problem arise in the last line of the code. It returns the parameter of the function. Figure 11 shows the accuracy of this conversion system with or without using the string. Here the graph describes the accuracy with the length of tokens. We can see that the system works perfectly for short length pseudo code line and the accuracy falls for the long length pseudo code.

F. MODEL ACCURACY

The accuracy of our model is evaluated by comparing the prediction rate of our model with the actual percentage. With a low precision and a high loss, the model would produce

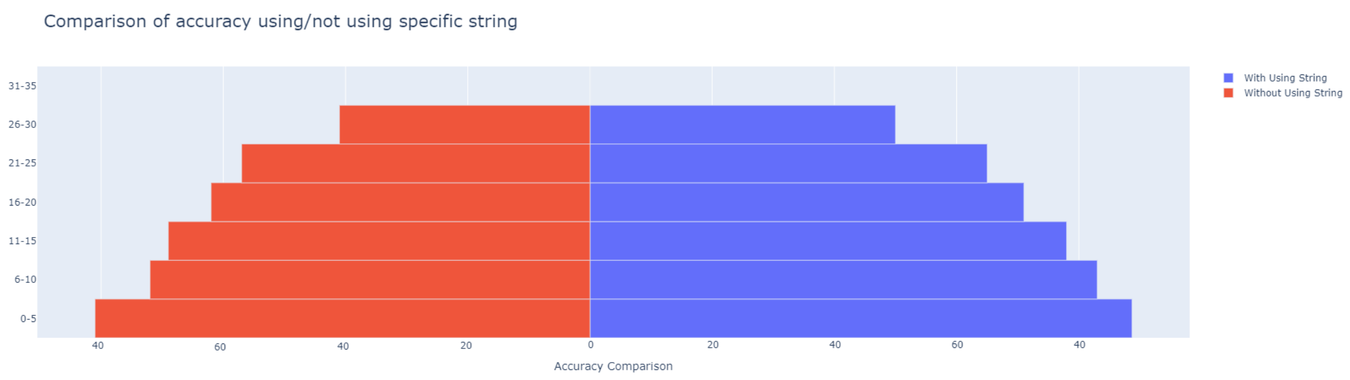


FIGURE 11. Comparison of accuracy with or without string.

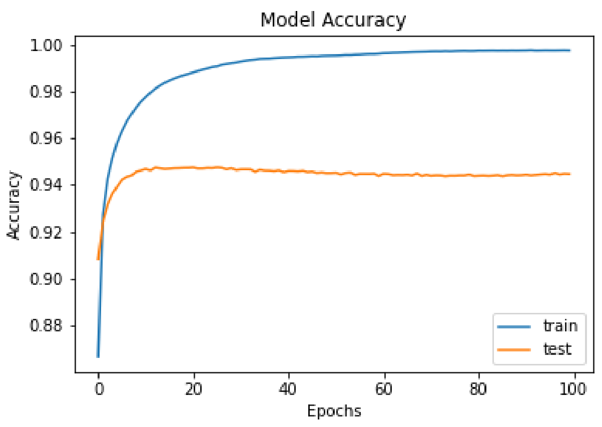


FIGURE 12. Accuracy of the proposed translation model.

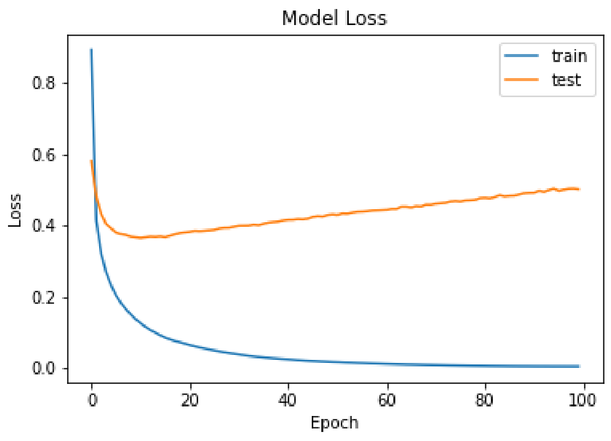


FIGURE 13. Loss of the proposed translation model.

many errors on most of the data. However, if both loss and precision are low, the model produces more minor errors. Nonetheless, it produces more errors in some data if the boss and precision both are high. Lastly, the ideal scenario is when there is high precision and a low loss, a few errors are produced.

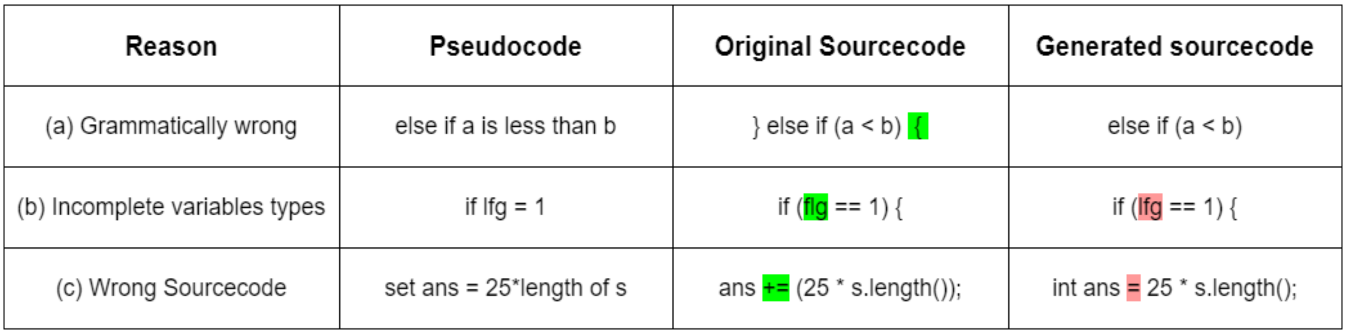
G. MODEL LOSS

Loss is the total number of errors produced by our model. Loss is a value to evaluate the performance of our model. When the errors are high, the loss is high, and the model does not perform well. The better our model performs, the lower loss it is. However, whether the loss is high or low, if we plot losses over time, we can evaluate the learning speed of our model as the model uses the loss function for learning in Seq2Seq. This approach is similar to gradient descent, which can modify the model's parameters using the information on the loss outcome.

We have organized the inaccurate output into three groups: (a) the source code is grammatically wrong, (b) sometimes, the source code contained incomplete variables in the case of logical expression, (c) a source code is incorrect. Figure 14 shows the classification of error analysis of this research.

Analysis of the transliteration results reveals the following:

1) Noticing all the output completely, we can see that the proposed approach can work if the pseudo-code statement's length and source code's length are different. Even if the output is not predicted correctly, the total input and output words could be different.
2) The seq2seq learning method successfully learns the semantic and syntactic relationships between the pseudo-code statement and source code. For this, the maximum output of the decoder is correct. The maximum outputs are logically correct. It is a good thing that the result is inspiring since converting the pseudo-code statement to source code is a complex task. The SPoC data set covered different types of pseudo-code statement. The suggested approach predicts such output since the seq2seq method is a state of the art deep learning model that can catch grammatical and logical rules diversity.
3) The seq2seq model can separate frequent and rare words in the pseudo-code statement. The frequent words are detected as code keyword, and rare words are detected as variables.
4) If the input pseudo-code statement is shorter, the seq2seq model predicts better output than the shorter ones which is shown in Table 5. However, the proposed model can also predict a longer pseudo code statement accurately.
5) For incorrect source code prediction, the approach predicts the word from the previous statement. If a rare



Reason	Pseudocode	Original Sourcecode	Generated sourcecode
(a) Grammatically wrong	else if a is less than b	} else if (a < b) {	else if (a < b)
(b) Incomplete variables types	if lfg = 1	if (flg == 1) {	if (lfg == 1) {
(c) Wrong Sourcecode	set ans = 25*length of s	ans += (25 * s.length());	int ans = 25 * s.length();

FIGURE 14. Classification of error analysis.

word is predicted, then the model detected that as a variable and declared it.

H. RESULT

The Bilingual Evaluation Understudy (BLEU) is a score that is calculated by comparing a candidate translation (source code) against one or more reference translations (pseudocode). In 2002, Papineni *et al.* [30] proposed this score to predict the accuracy using automatic machine translation systems. BLEU is not entirely effective though it has several interesting benefits. BLEU is faster and easier to calculate, highly interactive, language-independent and widely used. We used the BLEU score to determine the output source code. BLEU is computed using a couple of ngram modified precisions. Specifically,

$$BLEU = BP \,.\, exp\,(\sum_{n=1}^{N} w_n \,log\, p_n) \tag{12}$$

where, p_n is the modified precision for $n$gram, the base of log is the natural base e, w_n is weight between 0 and 1 for $log\ p_n$ and $\sum_{n=1}^{N} w_n = 1$, and BP is the brevity penalty to penalize short machine translations.

$$BP = \begin{cases} 1 & \text{if c > r} \\ exp\left(1 - \frac{r}{c}\right) & \text{if c} \le \text{r} \end{cases} \tag{13}$$

where, the number of unigrams (longitudinal) is c in the whole candidate sentences, and in the corpus, the best matching length is r. The closest reference penalty length to the candidate sentences is the best match length.

In general, BLEU is assessed on a corpus where numerous candidate phrases are translated and each of these phrases contains multiple reference phrases. c is the number of uni-grams in every sentence of the candidate, and r is the sum of the best matched lengths for each phrase on the corpus of the candidates. BLEU is always a value between 0 and 1 as BP, w_n,and p_n are always between 0 and 1, and

$$exp(\sum_{n=1}^{N} w_n \,log\, p_n) = \prod_{n=1}^{N} exp\,(w_n \,log\, p_n) \tag{14}$$

TABLE 6. Comparision of accuracy and error-rate between different types of model.

Method	Accuracy (%)	Error rate (%)
Search-based Process [13]	44.7	55.3
Semantic Scaffolds Method [15]	55.1	44.9
Natural Language Processing Approach [16]	72.8	27.2
Conceptual-Metamodel Mechanism [20]	76.2	23.3
Seq2Seq Process	88.7	11.3

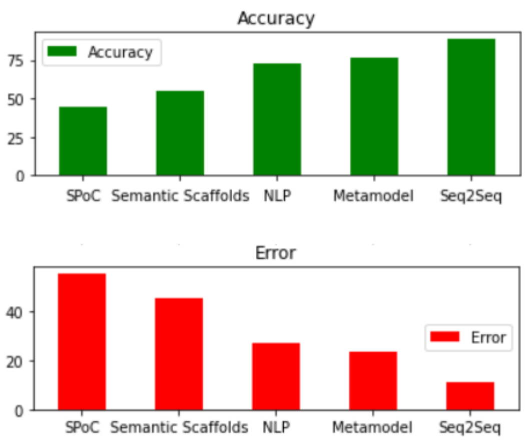


FIGURE 15. Comparison of accuracy of our models with existing models.

$$= \prod_{n=1}^{N} [exp\,(log\, p_n)]^{w_n} \tag{15}$$

$$= \prod_{n=1}^{N} p_n^{w_n} \tag{16}$$

$$\in [0, 1] \tag{17}$$

Usually, BLEU uses $N = 4$ and $w_n = \frac{1}{N}$.

The accuracy is calculated using the equation 18.

$$Accuracy = P \times 100\% \tag{18}$$

With the help of BLEU, the Error-rate is as equation 19.

$$Error\ Rate = (1 - P) \times 100\% \tag{19}$$

Other than Seq2Seq approach, different type of methods is considered for the conversion of pseudocode into source code. A comparison Table 6 shows the efficiency of source code and error rate for the proposed Semantic Scaffolds approach, NLP approach, and the Seq2Seq learning method considered.

Table 6 represents BLEU portion of efficiency for predicting the correct source code. The Seq2Seq learning method demonstrates an efficiency ratio of 88.7%. The proposed model is compared with other approaches on the SPoC dataset. The efficiency and error-rate estimate for the three models are represented in Figure 15.

V. CONCLUSION

The primary objective of this research work is to translate pseudo-code statements into appropriate source code. The main contribution of this research is that, we use neural machine translation for this task. We also solve the 26% blank pseudo-code problem of SPoC dataset. For text pre-processing, we use different types of NLTK library functions. The proposed model is heavily dependent on the sequence-to-sequence learning model.In order to tackle common natural language uncertainties, we have utilized text pre-processing. Natural language processing's primary work is to find out the clarification of pseudo-code instructions. We mapped out pseudo-code instruction into vectors of real numbers by word-to-vector model. Word embedding model predicted on the base of Skip-Gram architecture. The technique created is a conversion tool based on Seq2seq, since it uses NLP in this research in an AI way. We use the attention mechanism without mention that owns a significant influence on the results obtained at the end. Therefore, variant types of datasets will increase the accuracy of the proposed method. We can improve the projected architecture by using different types of LSTM networks or attention mechanism.

ACKNOWLEDGMENT

The authors would like to thank the help provided by the JnU-Emerging Technologies Research Lab of Jagannath University.

REFERENCES

[1] N. U. Koyluoglu, K. Ertas, and A. Brotman, "Pseudocode to code translation using transformers," Natural Lang. Process. With Deep Learn., Stanford, CA, USA, Tech. Rep. CS224N, 2021.

[2] T. Sharma, M. Kechagia, S. Georgiou, R. Tiwari, and F. Sarro, "A survey on machine learning techniques for source code analysis," 2021, *arXiv:2110.09610*.

[3] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," 2020, *arXiv:2006.03511*.

[4] B. Roziere, M. A. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 20601–20611.

[5] V. Parekh and D. Nilesh, "Pseudocode to source code translation," *Int. J. Emerg. Technol. Innov. Res.*, vol. 3, no. 11, pp. 45–52, 2016.

[6] Q. Zhu, W. Zhang, L. Zhou, and T. Liu, "Learning to start for sequence to sequence architecture," 2016, *arXiv:1608.05554*.

[7] M. Zavershynskyi, A. Skidanov, and I. Polosukhin, "NAPS: Natural program synthesis dataset," in *Proc. Workshop Neural Abstr. Mach. Program Induction (NAMPI)*, 2018. [Online]. Available: https://arxiv.org/abs/1807.03168

[8] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proc. 55th Annu. Meeting Assoc. Comput. Linguistics (ACL)*, Vancouver, BC, Canada, 2017. [Online]. Available: https://arxiv.org/abs/1704.01696

[9] M. Allamanis, D. Tarlow, A. Gordon, and Y. Wei, "Bimodal modelling of source code and natural language," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 2123–2132.

[10] S. Nadkarni, P. Panchmatia, T. Karwa, and S. Kurhade, "Semi natural language algorithm to programming language interpreter," in *Proc. Int. Conf. Adv. Hum. Mach. Interact. (HMI)*, Doddaballapur, India, Mar. 2016.

[11] S. Iyer, A. Cheung, and L. Zettlemoyer, "Learning programmatic idioms for scalable semantic parsing," 2019, *arXiv:1904.09086*.

[12] M. R. Amal, C. V. Jamsheedh, and L. S. Mathew, "Software tool for translating pseudocode to a programming language," *Int. J. Cybern. Informat.*, vol. 5, no. 2, pp. 79–87, Apr. 2016.

[13] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. Liang, "SPoC: Search-based pseudocode to code," 2019, *arXiv:1906.04908*.

[14] S. A. Hayati, R. Olivier, P. Avvaru, P. Yin, A. Tomasic, and G. Neubig, "Retrieval-based neural code generation," 2018, *arXiv:1808.10025*.

[15] R. Zhong, M. Stern, and D. Klein, "Semantic scaffolds for pseudocode-to-code generation," 2020, *arXiv:2005.05927*.

[16] A. T. Imam and A. J. Alnsour, "The use of natural language processing approach for converting pseudo code to C# code," *J. Intell. Syst.*, vol. 29, no. 1, pp. 1388–1407, 2019.

[17] G. Klein, Y. Kim, Y. Deng, J. Crego, J. Senellart, and A. M. Rush, "OpenNMT: Open-source toolkit for neural machine translation," 2017, *arXiv:1709.03815*.

[18] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," 2018, *arXiv:1808.09588*.

[19] A. Daza and A. Frank, "A sequence-to-sequence model for semantic role labeling," 2018, *arXiv:1807.03006*.

[20] T. Dirgahayu, S. N. Huda, Z. Zukhri, and C. I. Ratnasari, "Automatic translation from pseudocode to source code: A conceptual-metamodel approach," in *Proc. IEEE Int. Conf. Cybern. Comput. Intell. (CyberneticsCom)*, Nov. 2017, pp. 122–128.

[21] J. Klein, H. Levinson, and J. Marchetti, "Model-driven engineering: Automatic code generation and beyond," Dept. Softw. Eng., Carnegie Mellon Univ., Pittsburgh, PA, USA, 2015, pp. 67–72.

[22] V. Parekh and D. Nilesh, "Pseudocode to source code translation," *Int. J. Emerg. Technol. Innov. Res.*, vol. 3, no. 11, pp. 45–52, 2016.

[23] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 574–584.

[24] G. Yang, Y. Zhou, X. Chen, and C. Yu, "Fine-grained pseudo-code generation method via code feature extraction and transformer," 2021, *arXiv:2102.06360*.

[25] U. Ahmed, G. Srivastava, and J. C.-W. Lin, "Reliable customer analysis using federated learning and exploring deep-attention edge intelligence," *Future Gener. Comput. Syst.*, vol. 127, pp. 70–79, Feb. 2022.

[26] S. Abdelfattah, M. Baza, M. M. Badr, M. M. E. A. Mahmoud, G. Srivastava, F. Alsolami, and A. M. Ali, "Efficient search over encrypted medical data with known-Plaintext/Background models and unlinkability," *IEEE Access*, vol. 9, pp. 151129–151141, 2021.

[27] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, *arXiv:1301.3781*.

[28] M. Rosca and T. Breuel, "Sequence-to-sequence neural network models for transliteration," 2016, *arXiv:1610.09565*.

[29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.

[30] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. 40th Annu. Meeting Assoc. Comput. Linguistics (ACL)*, 2001, pp. 311–318.

UZZAL KUMAR ACHARJEE received the M.Sc. degree in computer science and the Ph.D. degree in applied physics, electronics & communication engineering from the University of Dhaka, Bangladesh, in 2000 and 2014, respectively. He is currently working as a Professor with the Department of Computer Science and Engineering, Jagannath University, Bangladesh. His research interests include the area of artificial intelligence, neural networks, deep learning, and data mining.

MINHAZUL AREFIN (Member, IEEE) was born in Narayangonj, Dhaka, Bangladesh, in 1995. He received the B.Sc. and M.Sc. degrees in computer science and engineering from Jagannath University, Dhaka. His research interests include machine learning, deep learning, digital image processing, artificial intelligence, data mining, machine translation, and natural language processing. He has been recognized as a Chairperson of IEEE JnU Student Branch and IEEE Computer Society JnU Student Branch, in 2018 and 2019, and got Extraordinary Volunteer Award, in 2018.

KAZI MOJAMMEL HOSSEN (Member, IEEE) received the B.Sc. degree in computer science and engineering from Jagannath University, Dhaka, in 2017, where he is currently pursuing the M.Sc. degree in computer science and engineering. He has published several good research articles in conference proceedings. His research interests include natural language processing, artificial intelligence, machine learning, and deep neural networks. He has recognized as a Secretary of IEEE JnU Student Branch, in 2018 and 2019.

MOHAMMED NASIR UDDIN received the Bachelor of Engineering degree in computer engineering from the Lviv Polytechnic National University, Lvov, Ukraine, the Master of Science degree in computer engineering, and the Ph.D. degree in computer science from the Moscow Power Engineering Institute, Technical University, Moscow, Russia. He is currently a Professor with the Department of Computer Science and Engineering, Jagannath University, Bangladesh. Prior to joining Jagannath University, he was a Faculty Member of the Department of Computer Science and Engineering, Moscow Power Engineering Institute, Technical University, from February 2001 to June 2010. He also worked as a Faculty Member of the Santo-Mariam University of Creative Technology and Uttara University, Bangladesh. His research interests include machine learning, deep learning, cyber security, and digital forensic.

MD. ASHRAF UDDIN received the B.Sc. and M.S. degrees in computer science and engineering from the University of Dhaka, Bangladesh, and the Ph.D. degree from the School of Engineering, IT, and Physical Sciences, Federation University, Australia. His research interests include the areas of blockchain, machine learning, security and privacy in remote patient monitoring, and modeling, analysis, and optimization of protocols in wireless sensor networks.

LINTA ISLAM received the B.Sc. and M.Sc. degrees in computer science and engineering from Jagannath University, Bangladesh. She is currently working as a Lecturer with Jagannath University. She does research in crowdsourcing, data science, and blockchain. She has published 20 papers in various journals and conferences. In her supervision, several papers have also been published in IEEE conferences.