mathematics

MDPI

Article

Retrieval-Based Transformer Pseudocode Generation

Anas Alokla [1], Walaa Gad [1], Waleed Nazih [2,*], Mustafa Aref [1] and Abdel-Badeeh Salem [1]

[1] Faculty of Computer and Information Sciences, Ain Shams University, Cairo 11566, Egypt; anas.hameed@cis.asu.edu.eg (A.A.); walaagad@cis.asu.edu.eg (W.G.); mostafa.aref@cis.asu.edu.eg (M.A.); absalem@cis.asu.edu.eg (A.-B.S.)
[2] College of Computer Engineering and Sciences, Prince Sattam Bin Abdulaziz University, Al-Kharj 11942, Saudi Arabia
* Correspondence: w.nazeeh@psau.edu.sa

Abstract: The comprehension of source code is very difficult, especially if the programmer is not familiar with the programming language. Pseudocode explains and describes code contents that are based on the semantic analysis and understanding of the source code. In this paper, a novel retrieval-based transformer pseudocode generation model is proposed. The proposed model adopts different retrieval similarity methods and neural machine translation to generate pseudocode. The proposed model handles words of low frequency and words that do not exist in the training dataset. It consists of three steps. First, we retrieve the sentences that are similar to the input sentence using different similarity methods. Second, pass the source code retrieved (input retrieved) to the deep learning model based on the transformer to generate the pseudocode retrieved. Third, the replacement process is performed to obtain the target pseudo code. The proposed model is evaluated using Django and SPoC datasets. The experiments show promising performance results compared to other language models of machine translation. It reaches 61.96 and 50.28 in terms of BLEU performance measures for Django and SPoC, respectively.

Keywords: natural language processing; retrieval-based; neural machine translation; pseudocode generation; deep learning-based transformer

check for updates

Citation: Alokla, A.; Gad, W.; Nazih, W.; Aref, M.; Salem, A.-B. Retrieval-Based Transformer Pseudocode Generation. *Mathematics* **2022**, *10*, 604. https://doi.org/10.3390/math10040604

Academic Editors: Nebojsa Bacanin, Catalin Stoean and Victor Mitrana

Received: 25 December 2021
Accepted: 13 February 2022
Published: 16 February 2022

1. Introduction

Converting source code to pseudocode is a sub-task of semantic analysis. This conversion is considered one of the problems of converting code to Natural Language (NL) descriptions [1–4]. It is a challenging problem because the input and the output are different in structure, syntax, and grammar. To solve this problem, there are several possible approaches to be followed. The use of neural networks [5–7] may solve this problem. However, this solution does not ensure that the results are structurally correct. The use of Machine Translation (MT) is the most widely used approach. There are several approaches to applying the MT such as Statistical Machine Translation (SMT) [3,8] and Neural Machine Translation (NMT) [9]. SMT is based on finding the alignment between the input and output sentences using statistical methods. This approach consumes time in the training process and its results are less than the NMT approach based on building two neural networks (i.e., encoder and decoder) [10,11]. Usually, there is an additional network to link the encoder and the decoder, this link is a Deep Neural Network (DNN) [12] or an attention layer [3].

Machine Translation (MT) has several structures in the input description and output form; sequence to sequence, sequence to tree, tree to sequence, and tree to tree. The application of such a structure depends on the nature of the input-output structure [13]. A sequence-to-tree is applied as it depends on converting the pseudocode to a source code. Moreover, the description of the source code can be represented in the structure of a tree. This representation has proven effective in improving results because it imposes a limited format on the source code. It is not an easy task and may lead to incorrect results with low-frequency tokens or tokens that do not exist in the training dataset. The main problem with NMT is that it depends on the Recurrent Neural Network (RNN) that takes a long time for training.

In this paper, a novel retrieval-based transformer pseudocode generation model is proposed to generate pseudocode from source code. The proposed model adopts DLBT [1] to overcome the problems of the training speed and the vanishing gradient [14]. In addition, the retrieval mechanism based on NMT [15] is added to the proposed model to deal with low-frequency tokens that do not exist in the training dataset.

Moreover, dynamic programming-based sentence-to-sentence alignment is applied to similar sentences for the word replacement procedure and enables the retrieval of incomplete matches. The retrieval-based transformer pseudocode generation model consists of:

- Retrieval: Finding the most similar sentence between the input and the training set.
- Deep learning-based transformer: passing the retrieved code (i.e., retrieved input) to the encoder-decoder transformer.
- Replacement process: by using information from input, retrieved input, and retrieved output (i.e., retrieved pseudocode).

These contributions allow us to improve results in terms of BLEU performance measures compared to other Machine Translation (MT) methods. Two different datasets are used to evaluate the retrieval-based transformer model, the highest improvement is 61.96 in terms of the BLEU measure.

This paper is organized as follows: Section 2 presents the related work, Section 3 presents the proposed model, Section 4 shows experimental results, and finally, Section 5 the conclusion.

2. Related Work

Machine Translation (MT) is of great importance as it is not only a process of linguistic translation from one language to another, but there are also multiple uses for it. These uses include: summarizing linguistic texts, finding descriptions of the programming methods by summarizing the target of the method, and converting the Natural Language (NL) (pseudocode) into a source code and vice versa. There are many ways to apply Machine Translation (MT) such as Rules-Based Machine Translation (RBMT) and Statistical Machine Translation (SMT). Recently, the use of NMT in linguistic modeling has increased due to its promising results.

A Deep Learning Model Based on a Transformer (DLBT) [1] was proposed to convert the source code into pseudocode. This system has three components; tokenization and embedding, transformer, and post-processing. Tokenization converts every sentence of the code to a set of tokens and embedding assigns similar encodings to similar tokens. Using the transformer is a good choice to avoid problems related to weight. In addition, the transformer processes the whole sentence in the training phase, which saves the required processing time. Moreover, the transformer shows promising results in language translation [16] and shows the same level in the translation process from the source code of the pseudocode. The last component (i.e., post-processing) is based on handling minor errors that occur due to the encoding process and the translation process, such as leaving blanks. This system cannot handle the tokens that do not exist or the tokens with low-frequency in the training dataset.

Yang et al. [2] generated the pseudocode in three phases. In the first phase (i.e., the encoding phase), a transformer was utilized to encode the source code. Then, local features were extracted from the source code using Convolutional Neural Network (CNN). In the third phase, in the decoding phase, a generator was implemented to generate the corresponding pseudocode using beam search. Moreover, they used the projected gradient descent method to add adversarial data to the embedding layer training to make the model more robust and avoid overfitting. Finally, they tested many attention mechanisms and studied their effect on the results. The limitation of this approach is the same as the limitation of [1].

Alhefdhi et al. [3] presented a model for converting the source code to pseudocode using NMT based on the RNN. They used Long-Short-Term Memory (LSTM) as the main unit in both encoder and decoder. LSTM is very important to avoid the vanishing gradient problem [14]. In addition, they used an attention layer to link between the encoder and decoder; this layer improves the results and has an important role in the alignment of input and output sentences. The training phase of this model consumes time since it processes the source code word by word. Moreover, the LSTM takes up a large amount of memory which needs more resources to complete the training process.

In [4] authors present a model for converting source code to pseudocode using Statistical Machine Translation (SMT). The proposed model includes two machine translations: Phrase-Based Machine Translation (PBMT) and Tree-to-String Machine Translation (T2SMT). PBMT alignments between the input and output while T2SMT transforms the input to tree using Abstract Syntax Trees (AST) [17] to preserve its structural context. This model has two limitations; many translation errors in the case of sentences that do not have enough similar sentences in the training dataset and SMT has also been shown to be less efficient than NMT [10,11].

Deng et al. [18] proposed a model to generate pseudocode from source code. This model begins with the input encoder to encode the embedded vector of all input data into a latent vector space using Bidirectional LSTM (BiLSTM). Then, a sketch decoder performs as a connecting medium between the input encoder and the natural language explanation decoder. The LSTM and the attention layer are used to achieve the sketch decoder purpose. Finally, the natural language explanation decoder decodes and generates the natural language using an RNN followed by the attention mechanism and copy mechanism [19]. The limitation of this approach is the same as [3].

Handling low-frequency tokens and tokens that do not exist in the training dataset was proposed in [20] using the retrieval mechanism [15] and the model proposed in [12]. The proposed model has four steps to generate the source code. Firstly, using the input sentence to search for the most similar sentence to it in the training dataset. Secondly, n-grams were extracted from the output sentence of the first step and composed an AST as a partial tree. Thirdly, the similar tokens are searched between the output and the retrieved sentence and then replaced with the tokens that are not similar between the input sentence and the retrieved sentence in the first step. Fourthly, update the weights to increase the probability at each decoding step. This model is more flexible, especially if the training dataset is not huge which improves the results, but it has the same limitation as [3].

In [21], the authors proposed a model for program detection and classification by the function of the source code using a Tree-based Convolutional Neural Network (TBCNN). This model begins with an input layer in which it enters the source code after converting the source code into a tree using AST and then transforming nodes into vectors. The second layer is the max-pooling layer for dynamic pooling generation. The third layer is fully connected to connect all the neurons to a single linear layer. The last layer, SoftMax, is for defining and classifying the output. The limitation of this approach is that it works over a fixed-size sentence. Furthermore, performance may decrease if the sentence becomes too long.

DeepCom was proposed in [22], it is a comment generator for Java methods. This model starts with data processing for files that contain the source code of Java methods and the comments represented in a natural language. Then, the source code is converted to AST sequences. In the second step, an NMT was trained using the previously created ASTs. This structure of this NMT is an RNN and its unit is LSTM in both the encoder and decoder. The third step is to generate comments after the training process by passing the source code of the Java method to NMT to generate the comment. The limitation of this approach is the same as [3], in addition to not using the attention layer may enhance the performance. Later in [23], they presented a modified version of DeepCom which outperforms the previous one. The new model uses AST sequences and the source code to create the required methods comments. In addition, it uses the beam search instead of creating comments word by word.

In Rule-Base system proposed in [24], authors extract more information from the source code, the system converted the Python code to XML code. Then, this information was used to generate the pseudocode using predefined templates. They analyzed the dataset of [4] to design the required rules to convert source code to pseudocode and to create the required templates. Converting the code to XML is an extra step that needs extra resources. Furthermore, the designed rules and templates are dataset dependent.

Zhang et al. [25] tried to handle the words with low frequency in the dataset. They suggested a source code summarization system that integrated retrieval-based methods with NMT-based methods. In the offline training phase, an attention encoder-decoder model was trained. Later, in the online testing phase, the system syntactically and semantically chose the best two similar code segments and passed them to the encoder, then, the decoder generates the summary by fusing the previous two segments. Finally, they evaluated the proposed systems using some metrics such as BLEU and hired more than one hundred workers to evaluate the system performance.

3. Retrieval-Based Transformer Pseudocode Generation

In this work, the retrieval-based transformer pseudocode generation model is proposed. The proposed model used the encoder-decoder transformer structures as in [1] and integrates the retrieval mechanisms for better pseudocode code generation. Moreover, it adapts the similarity methods for the retrieved code and the translation pieces [15]. Figure 1 shows the framework of the proposed retrieval-based transformer generation model.

The proposed model consists of three phases as follows:

1. Retrieval: Finding the most similar sentence between the input and training set.
2. Deep learning-based transformer: passing the retrieved code (i.e., retrieved input) to the encoder-decoder transformer.
3. Replacement process: by using information from input, retrieved input, and retrieved output (i.e., retrieved pseudocode).

Figure 2 shows an example for testing the retrieval-based transformer generation model.

3.1. Similarity between Input and Training Dataset (Retrieval Phase)

In this phase, many similarity methods are adapted to obtain the similarity between input and output. In testing time, the input source code X is passed to the similarity method. The training dataset is searched for a sentence similar to the input sentence X in the form of tokens to obtain the retrieved sentence X^m. There are different methods for calculating similarity and obtaining the retrieved sentence such as Latent Semantic Indexing (LSI) [26], Vector Space Model (VSM) [27], Nearest Neighbor Generator (NNGen) [28], and Levenshtein [29]. The LSI is an NLP technique for obtaining the text retrieval by analyzing the latent meaning or documents concepts. The VSM is an algebraic model that deploys Term Frequency-Inverse Document Frequency (TF-IDF) for representing text documents as vectors of identifiers. The NNGen is an algorithm used as a term frequency for word changes and returns the nearest neighbors of word changes using cosine similarity of vectors and the BLEU-4 score [30]. Levenshtein is an algorithm that is used for computing the edit distance between two sentences or words. Calculating similarity using Levenshtein utilizes the following equation [15,19]:

$$sim(X, X^m) = 1 - \frac{d(X, X^m)}{\max(|X|, |X^m|)} \tag{1}$$

where the $d(X, X^m)$ is the distance between the input X (i.e., source code) and the retrieved sentence is computed by using Levenshtein. In the proposed model, the LSI, SVM, NNGen, and Levenshtein are used to retrieve the highest similar M sentences from the training dataset where M is a hyperparameter that indicates the max retrieved sentences.

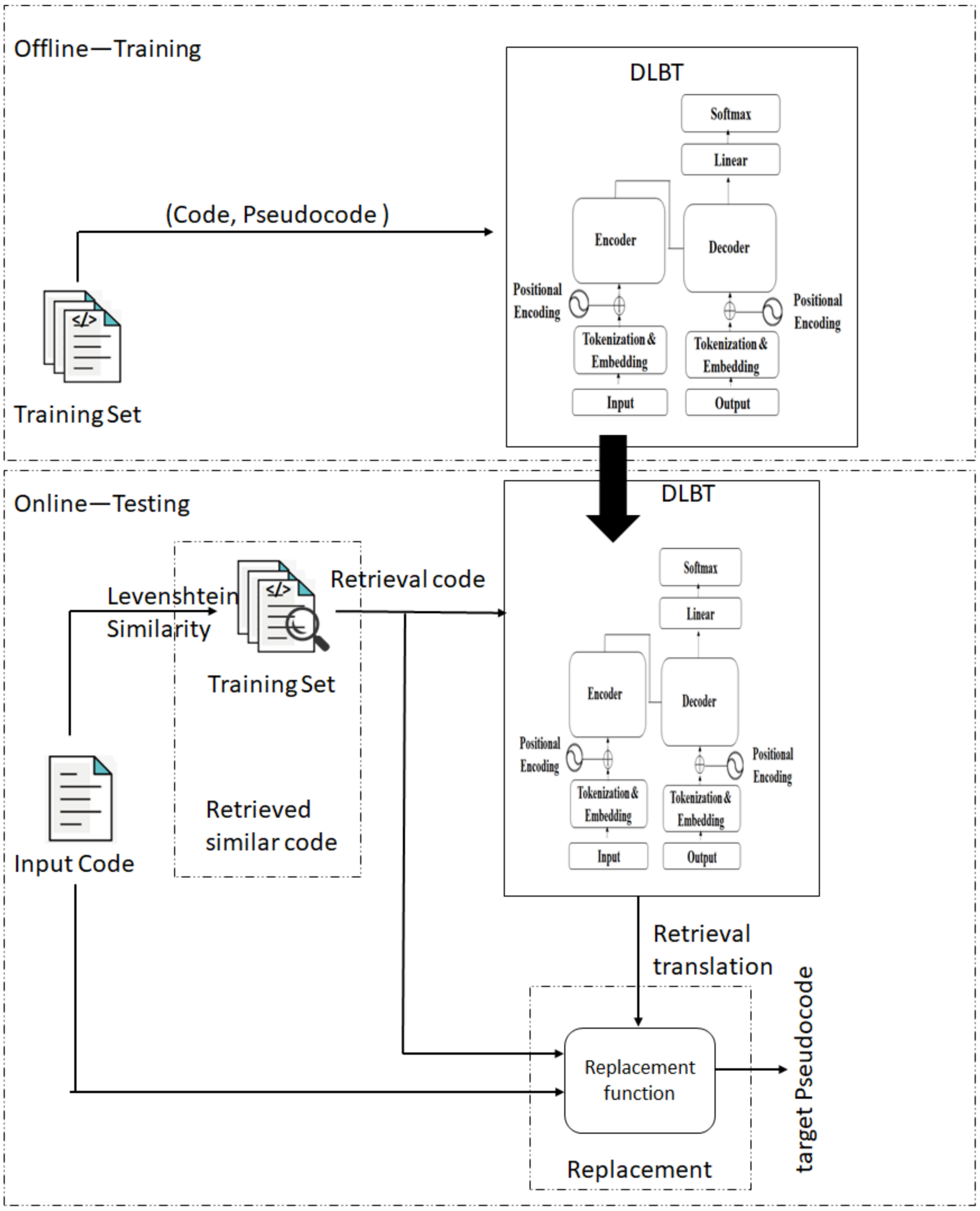
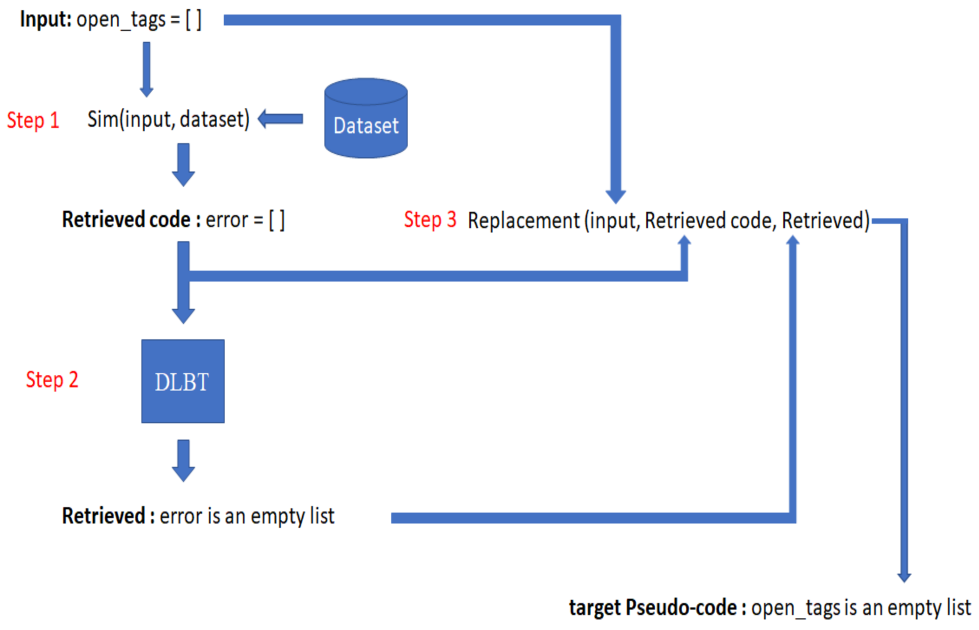


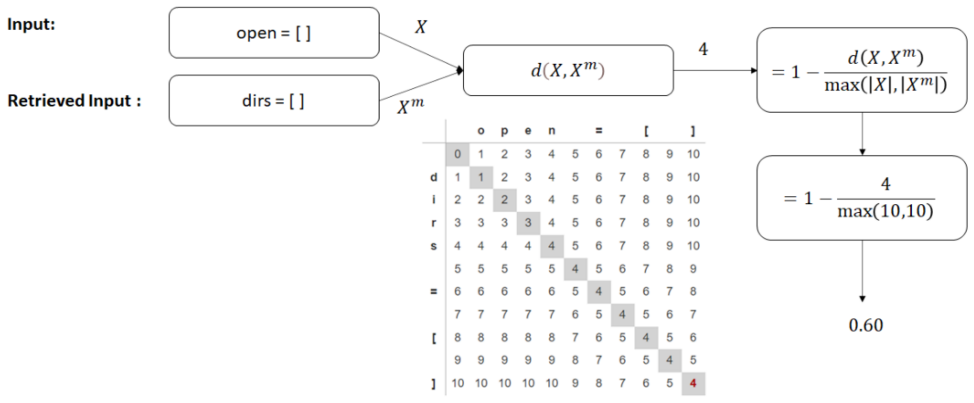
Figure 1. The framework of the proposed retrieval-based transformer generation model.



Input: open_tags = [ ]
Step 1
Sim(input, dataset)
Dataset
Retrieved code : error = [ ]
Step 3
Replacement (input, Retrieved code, Retrieved)
Step 2
DLBT
Retrieved : error is an empty list
target Pseudo-code : open_tags is an empty list

Figure 2. An example for retrieval-based transformer generation model from source code as an input to pseudocode as an output.

Figure 3 presents similarity calculations between input sentence "open = [ ]" and retrieved sentence "dirs = [ ]" using the Levenshtein method.



Input:
open = [ ]
Retrieved Input :
dirs = [ ]
X
X^m
$d(X, X^m)$
4
$= 1 - \frac{d(X, X^m)}{\max(|X|, |X^m|)}$
$= 1 - \frac{4}{\max(10,10)}$
0.60

		o	p	e	n		=		[		]
	0	1	2	3	4	5	6	7	8	9	10
d	1	1	2	3	4	5	6	7	8	9	10
i	2	2	2	3	4	5	6	7	8	9	10
r	3	3	3	3	4	5	6	7	8	9	10
s	4	4	4	4	4	5	6	7	8	9	10
	5	5	5	5	5	4	5	6	7	8	9
=	6	6	6	6	6	5	4	5	6	7	8
	7	7	7	7	7	6	5	4	5	6	7
[	8	8	8	8	8	7	6	5	4	5	6
	9	9	9	9	9	8	7	6	5	4	5
]	10	10	10	10	10	9	8	7	6	5	4

Figure 3. Calculating similarity between two sentences using Levenshtein distance.

The next formula was used for finding all retrieved sentences of the input training dataset [15]:

$$sim(X, D_{training}) = \underset{1 \le m \le M}{Max} sim(X, X^m) \tag{2}$$

M is a hyperparameter that indicates the maximum number of retrieved sentences. $X^m : \{1 \le m \le M\}$.

After obtaining the retrieved sentences for the input, similarity values and index of each sentence saved in a vector, then we sort this vector and take the top of it to be M. In Figure 4, an example is presented to obtain the most similar sentences with input "open = [ ]" and M equals 3.

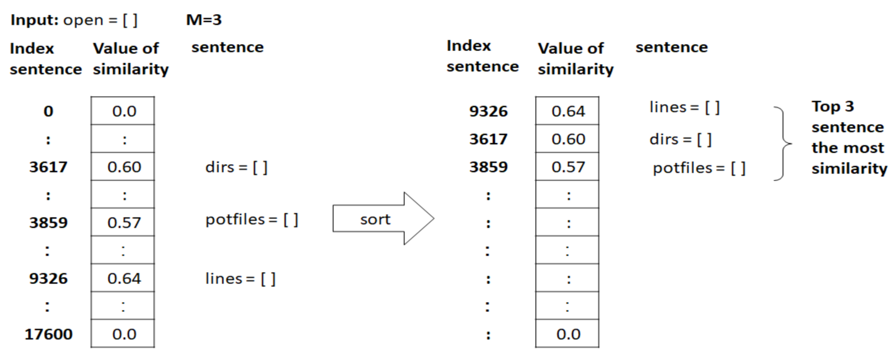


Figure 4. An example of retrieving the maximum of the most similar sentences (i.e., M).

3.2. *Deep Learning-Based Transformer*

In this step, DLBT is used as in [1]. Each retrieved sentence (i.e., retrieved code) X^m is passed to DLBT for generating the retrieved pseudocode Y^m that corresponds to the retrieved sentence X^m. The pseudocode is generated in three steps: tokenization and embedding, transformer, and post-processing. In the first step, each source code sentence and its equivalent pseudocode are tokenized and two special tokens (i.e., <sop> and <eop>) are added for each sentence to indicate the start and end of it. Then, embedding was performed to assign similar encodings for similar tokens. In the second step, a transformer was trained for matching between input source code sentence and output pseudocode sentence in the training phase. This transformer has an encoder and a decoder that have the same number of chained layers. The encoder processes the input, and the decoder processes the result of the encoder and the output pseudocode. Both encoder and decoder use the attention mechanism for weighs [1]. Finally, tokenization errors such as extra spaces were corrected in the post-processing step.

3.3. *Replacement for Generating the Target Pseudocode*

To generate pseudocode Y, it is necessary to take into account the original input code (X), the retrieved code (X^m) which is the most similar to the input code, and the retrieved pseudocode from DLBT (Y^m). The replacement process begins with obtaining the word alignments between X^m and Y^m and recording it in A^m. Second, recode the unedited words of X^m in W^m. Third, collect n-grams (up to 4 g) from retrieved pseudocode Y^m in G_X^m using A^m and W^m, where G_X^m indicates the possible translation pieces of X. Fourth, use the word-level alignments to select n-grams related to X and discard n-grams that are not related to X.

For example, as shown in Figure 5, the retrieval of input " open_tags = [ ]" is "error = [ ]" where the red token presents the unedited tokens (i.e., words) between input and the retrieved input. The blue part in the retrieved translation is collected as translation pieces of the input sentence. The token "=" is split into " is" and "an" and the token "[ ]" is split into " empty" and "list". The green part in target translation is the replacement piece of the input sentence.

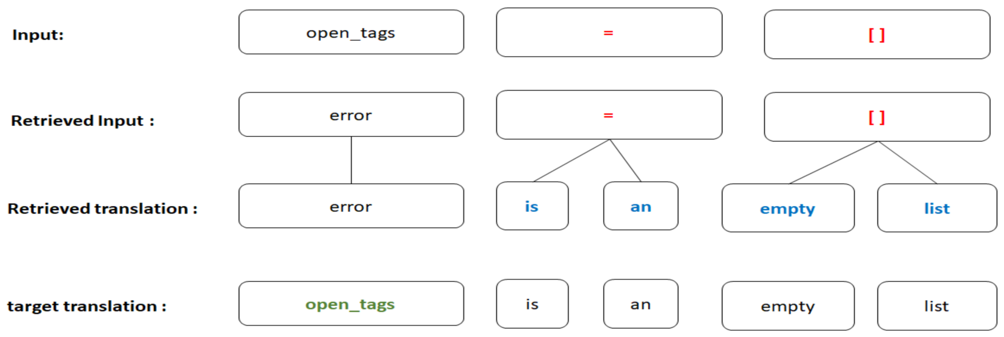
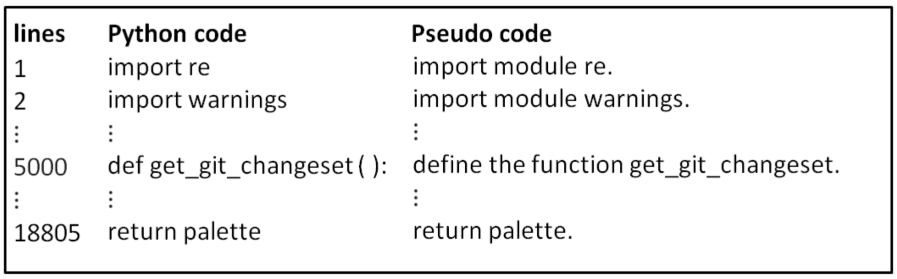


Figure 5. An example that presents the unedited tokens, translation pieces, and replacement pieces.

4. Experimental Results and Evaluation

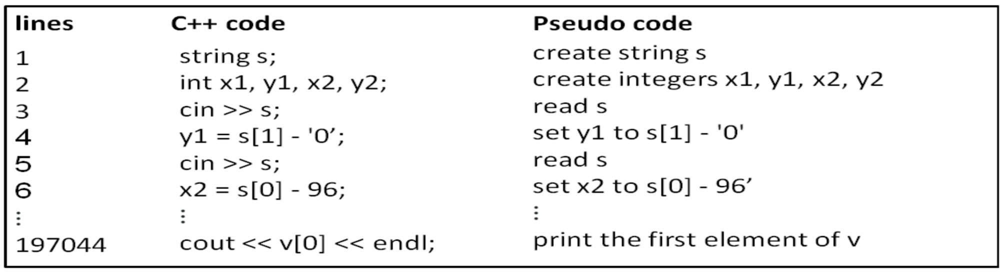
4.1. Dataset Description

Django dataset [3] and SPoC dataset [31] were used to train the proposed model. The source code of the Django dataset is written in Python while the source code of SPoC is C++. Both of the datasets have the corresponding pseudocode written in English. An example of the Django and SPoC datasets is shown in Figures 6 and 7.



lines	Python code	Pseudo code
1	import re	import module re.
2	import warnings	import module warnings.
⋮	⋮	⋮
5000	def get_git_changeset ( ):	define the function get_git_changeset.
⋮	⋮	⋮
18805	return palette	return palette.

Figure 6. An example of Django dataset.



lines	C++ code	Pseudo code
1	string s;	create string s
2	int x1, y1, x2, y2;	create integers x1, y1, x2, y2
3	cin >> s;	read s
4	y1 = s[1] - '0';	set y1 to s[1] - '0'
5	cin >> s;	read s
6	x2 = s[0] - 96;	set x2 to s[0] - 96'
⋮	⋮	⋮
197044	cout << v[0] << endl;	print the first element of v

Figure 7. An example of SPoC dataset.

In the SPoC dataset, the same source code may have a different pseudocode description as shown in Table 1. This affects the training phase since it disturbs the appropriate words and the corresponding source sentence. In addition, it affects the experimental results in testing because even the output is a correct sentence, it may not be considered correct in the evaluation process as the corresponding sentence may be another synonym translation. To avoid this problem, we used only the first pseudocode for the same source code.

Table 1. An example of source code has many corresponding pseudocodes.

SPoC Dataset	
C++ Code	**Pseudocode**
string *s*;	create string *s* declare string variable *s* declare *s* as string *s* = string
*x*1 = *s*[0]-96;	set *x*1 to *s*[0]-96 set *x*1 = *s*[0]-96 *x*1 = *s*[0]-96

if (*x*2 > *x*1)	if *x*2 is greater than *x*1
	if *x*2 > *x*1

We split each dataset into training and testing. Table 2 presents the training and testing samples for Django and SPoC datasets. PyTorch [32] and Spacy [33] were used for implementing the proposed model. In addition, WordNet [34] was used for converting source code into tokens.

Table 2. The number of training and testing samples for Django and SPoC datasets.

Dataset	Training	Testing
Django	17,605	1200
SPoC	181,862	15,183

4.2. Performance Evaluation

For evaluating the used machine translation techniques, we use the BLEU score metric. This metric measures the performance quality of the machine translation output by matching between the translation output (candidate or hypothesis) and human translation output (reference) as shown in Equation 3. The value of the BLEU score metric is between 0 to 1 in which the 1 value means that the candidate matches completely with reference. The 0 value means no matching between candidate and reference. There are different forms of BLEU metrics according to the difference grams such as uni-gram and bi-gram. The precision score is defined as follows:

$$P_n = \frac{\sum_{ngram \epsilon C} count_{clip}(ngram)}{\sum_{ngram' \epsilon C} count(ngram')} \tag{3}$$

where $\sum\limits_{ngram \epsilon C} count_{clip}(ngram)$ is the n-gram matches sentence by sentence, and $\sum\limits_{ngram' \epsilon C} count(ngram')$ is the number of candidate n-grams in the testing dataset.

BP is the brevity penalty is defined as follows:

$$BP(r,c) = \left\{ \begin{array}{ll} e^{(1-\frac{|r|}{|c|})} & if\ |r| \geq |c| \\ 1 & otherwise \end{array} \right\} \tag{4}$$

where r is the reference sentence, $|r|$ is the length of the reference sentence in the dataset, c is the candidate sentence, and $|c|$ is the length of candidate translation.

BLEU is defined as follows:

$$BLEU(r,c) = BP(r,c).\exp(\sum_{n=1}^{N} w_n.\log P_n) \tag{5}$$

Using n-grams up to length N and positive weights $\sum_{n=1}^{N} w_n$ summing to one. The calculating of accuracy by summation of every BLEU score value equals one and is divided by the number of testing sentences. Since the proposed model generates a list of candidate target pseudocode between 1 and M. The maximum value from candidate target and reference pseudocodes was used.

$$BLEU = {}_{1\leq t\leq M}^{Max} BLEU\left(Y^a, Y^t\right) \tag{6}$$

where the Y^a indicates the reference pseudocode and Y^t is the target pseudocode.

4.3. Results

In the retrieval phase, many similarity methods are applied to find a sentence similar to the input sentence X in the form of tokens to obtain the retrieved sentence X^m. Such similarity methods calculate similarity and obtain the retrieved sentence such as Latent Semantic Indexing (LSI) [26], Vector Space Model (VSM) [27], Nearest Neighbor Generator (NNGen) [28], and Levenshtein [29]. Sometimes, the same source code may have a different pseudocode description as shown in Table 1. The proposed model uses only the first pseudocode for the same source code. Table 3 presents the results improvements of different retrieval methods after using the first pseudocode in case we have many pseudocodes for the same source code. Levenshtein similarity method records 38.15, 17.56, 60.40, and 49.2 in terms of BLEU and Accuracy for Django, SPoC datasets, respectively.

Table 3. Improvement of retrieval methods after using the first pseudocode of source code that has many pseudocodes in the dataset in terms of BLUE, accuracy, and time.

Method	**Django Dataset**					**SPoC Dataset**				
	BLEU		**Accuracy**		**Time**	**BLEU**		**Accuracy**		**Time**
	All	**First**	**All**	**First**	**-**	**All**	**First**	**All**	**First**	**-**
VSM	28.43	30.59	12.99	15.93	0.08 s	14.54	20.50	1.66	3.17	0.24 s
LSI	28.61	30.81	12.92	15.73	7.75 s	14.62	20.68	1.64	3.12	79.90 s
NNGen	30.41	31.66	13.97	16.62	0.0025 s	27.44	38.58	8.08	26.48	0.009s
Levenshtein	34.79	38.15	12.97	17.56	0.066 s	38.85	60.40	16.80	49.205	0.12 s

The second step in the proposed retrieval-based transformer pseudocode generation model, DLBT is used as in [1] to generate the retrieved pseudocode, which is an encoder-decoder structure. Experiments are conducted using 6 and 8 layers for the encoder-decoder structure. Table 4 has a sample of Python code and the corresponding pseudocode in 5 forms: written by a professional programmer, generated by DLBT using 6 and 8 layers, and generated by the proposed model using 6 and 8 layers. Tables 5 and 6 shows the performance of the proposed model for Django and SPoC datasets

Table 4. Retrieval-based transformer pseudocode generation model (SPoC dataset).

#	Python Code	Pseudocode
	Manually Generated Pseudocode	
1	from io import BytesIO	#from io import BytesIO into default name space.
2	open_tags = [ ]	#open_tags is an empty list.
3	if six. PY2	# if six.PY2 is true,
4	for supported_code in _supported:	# for every supported_code in _supported,
5	s = s. replace ('\t', '\\t')	#replace every occurrence of '\t' in s with '\\t'.
	DLBT (6-layers)	
1	from io import BytesIO	# from io import BytesIO into default name space.
2	open_tags = [ ]	#Substitute cache.get for cache_get.
3	if six. PY2:	# If true,Six.py2 is true.
4	for supported_code in _supported:	# For every supported in this value,
5	s = s. replace ('\t', '\\t')	#s is an empty list. Replace every occurrence of '\t' 'in twith '\\t'.
	DLBT (8-layers)	
1	from io import BytesIO	# from io import BytesIO into default name space.
2	open_tags = [ ]	##Substitute pattern.regex.pattern for template_dir.
3	if six. PY2:	# If six.py2 is true, If six.py2 is true,
4	for supported_code in _supported:	# For every template_dir in supported,
5	s = s. replace ('\t', '\\t')	#replace all occurrences of 't 'with 't '.
	Retrieval-based Transformer Generation (6-layers)	
1	from io import BytesIO	# from io import BytesIO into default name space.
2	open_tags = [ ]	#open_tags is an empty list.
3	if six. PY2:	# if six.PY2 is true,
4	for supported_code in _supported:	# for every supported_code in _supported
5	s = s. replace ('\t', '\\t')	#s is an empty list. Replace every occurrence of '\t' 'in s with '\\t'.
	Retrieval-based Transformer (8-layers)	
1	from io import BytesIO	# from io import BytesIO into default name space.
2	open_tags = [ ]	#open_tags is an empty list.
3	if six. PY2:	# if six.PY2 is true,
4	for supported_code in _supported:	#for every supported_code in _supported
5	s = s. replace ('\t', '\\t')	#replace every occurrences of '\t' in s with '\\t'.

Table 5. Retrieval-based transformer pseudocode generation model (Django dataset).

#	C++ Code	Pseudocode
	Manually Generated Pseudocode	
1	int n;	create integer n
2	cin >> n;	read n
3	for (int i = 0; i < (int)ans.size(); i++) {	for i = 0 to integer casted size of ans exclusive
4	if (x == cur) {	if x is equal to cur
5	for (int i = 0; i < a; i++) { cin >> a[i]; }	for i = 0 to a exclusive, read a[i]
	DLBT (6 layers)	
1	int n;	create integer n
2	cin >> n;	read n
3	for (int i = 0; i < (int)ans.size(); i++) {	for i = 0 to size of aexclusive
4	if (x == cur) {	if x is equal to 0
5	for (int i = 0; i < n; i++) { cin >> a[i]; }	for i = 0 to nexclusive, thenread a[i]

Table 5. *Cont.*

#	C++ Code	Pseudocode
	Retrieval-based Transformer (6 layers)	
1	int n;	create integer n
2	cin >> n;	read n
3	for (int i = 0; i < (int)ans.size(); i++) {	for i = 0 to size of ans exclusive
4	if (x == cur) {	if x is equal to cur
5	for (int i = 0; i < n; i++) { cin >> a[i]; }	for i = 0 to a exclusive, read a[i]

Table 6. Retrieval-based transformer performance in terms of BLEU and accuracy measures.

Dataset	Similarity Methods							
	LSI		VSM		NNGen		Levenshtein	
	BLEU	Acc	BLEU	Acc	BLEU	Acc	BLEU	Acc
Django (6-layers)	55.96	32.16	55.23	31.52	59.74	37.17	**61.96**	**40.81**
Django (8-layers)	53.14	31.76	52.32	30.81	57.63	34.07	60.55	38.94
SPoC (6-layers)	22.31	3.02	22.03	3.17	40.79	27.14	**50.28**	**34.72**
SPoC (8-layers)	20.83	2.05	20.67	2.31	39.43	25.74	**45.37**	**27.94**

Using the Django dataset, both DLBT and the proposed retrieval-based transformer generation model generate the correct pseudocode for line 1. In line 2, both DLBT with 6 layers and 8 layers generate the incorrect pseudocode. The result is not similar to any of the codes, because the "open_tags" token does not exist in the training dataset. Thus, there is a distortion in the prediction process. In the case of line 3, DLBT with 6 layers adds "true" after "if" and with 8 layers it duplicates "if six.PY2 is true,". On the other hand, the same instance model with the proposed retrieval-based transformer generation model retrieval-based mechanism corrects this error as it retrieved the similar input and applies the retrieval-based mechanism. In line 4, DLBT with 6 layers translates the token "supported_code" to "supported", because the token "supported_code" does not exist in the training dataset. The token "_supported "was translated to " this value " because the token "_supported " is very rare. DLBT with 8 layers translates the token "supported_code" to " template_dir" for the same previous reason. In line 5, the DLBT with 6 layers adds "s is an empty list" in both lines. In addition, DLBT with 6 and 8 layers do not generate "\\t" and "\t", because these tokens are not in the training set.

Finally, DLBT with 8 layers changes "every" by "all" in line 5, but the proposed model with 8 layers correct this problem because the retrieval-based mechanism retrieved the similar input and replaced token with correct input tokens. For the SPoC dataset, the proposed model with 6 layers translates "ans" correctly. In Table 5 the DLBT with 6 layers adds "then" in the generated pseudocode for line 5 because some pseudocodes have "then" and other similar pseudocodes do not have "then". The proposed retrieval-based transformer generates this token correctly.

In the next experiment, we compare the proposed retrieval-based transformer and DLBT in terms of accuracy and BLEU measures using Django and SPoC datasets. As shown in Table 6, a retrieval-based transformer with 6 layers achieved 40.81 accuracy and 61.96 BLEU over Django. In addition, with 8 layers it achieved 61.55 BLEU measure and 38.94 accuracy. In the case of the SPoC dataset, the retrieval-based transformer with 6 layers achieved 34.72 accuracy and 50.28 BLEU measure which outperforms the DLBT. Table 7 shows the BLUE measure of the proposed retrieval-based transformer performance compared to other state-of-the-art systems such as DLBT [1] and DeepPseudo [2]. Figures 8 and 9 show how the deep learning-based transformer (6 layers) performs during the training and testing phases to explain there is no overfitting.

Table 7. Retrieval-based transformer performance compared to other state-of-the-art systems.

Dataset	Model	BLEU
Django	Retrieval-based Transformer (6 layers)	**61.96**
	Retrieval-based Transformer (8 layers)	**61.29**
	DLBT (6 layers and without cross-validation)	59.62
	DLBT (8 layers and without cross-validation)	58.58
	Code2NL [18]	56.54
	code2pseudocode [3]	54.78
	T2SMT [4]	54.08
	DeepPseudo [2]	50.81
	Code-GRU [18]	50.81
	Seq2Seq w Atten. [2]	43.96
	NoAtt [3]	43.55
	RBMT [21]	41.87
	CODE-NN [2,18]	40.51
	ConvS2S [2]	37.45
	Seq2Seq w/o Atten. [2]	36.48
	Seq2Seq [18]	28.26
	PBMT [3]	25.17
	SimpleRNN [3]	06.45
SPoC	Retrieval-based Transformer (6 layers)	**50.28**
	DLBT (6 layers and without cross-validation)	48.12
	DeepPseudo [2]	46.45
	Retrieval-based Transformer (8 layers)	45.37
	Transformer [2]	43.73
	DLBT (8 layers and without cross-validation)	43.16
	Seq2Seq w Atten. [2]	41.00
	ConvS2S [2]	34.19
	Seq2Seq w/o Atten. [2]	33.76
	CODE-NN [2,18]	32.10

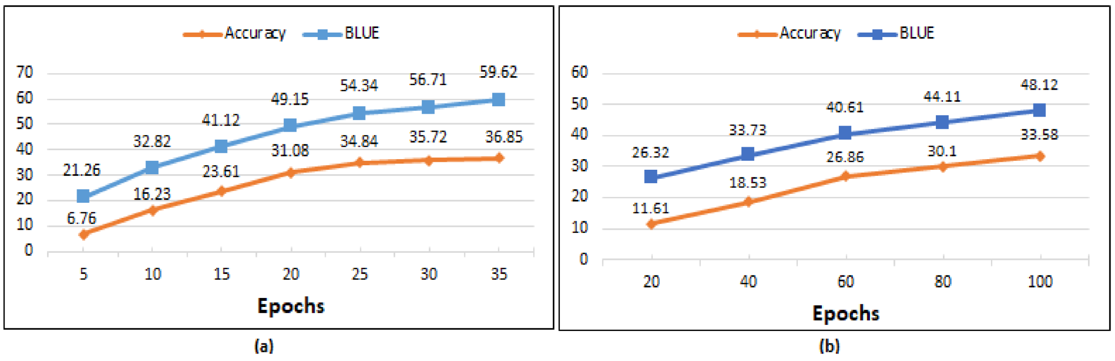


Figure 8. Transformer performance in test phase for (**a**) Django dataset and (**b**) SPoC dataset.

The proposed model using 6 layers records better performance due to:

- In a deep learning-based transformer, DLBT gives better performance using six layers.
- In the retrieval process, some of the pseudocodes generated using 8-layer DLBT have tokens which “Replace function” fails to obtain the proper replacement. For example, “if space == 0:” converted to “If space equals to string ‘0 ‘,” while the actual pseudocode “if space equals integer 0,”. Tokens “to string ‘ ‘“ do not refer to the input and cannot be replaced with “Replace function”.

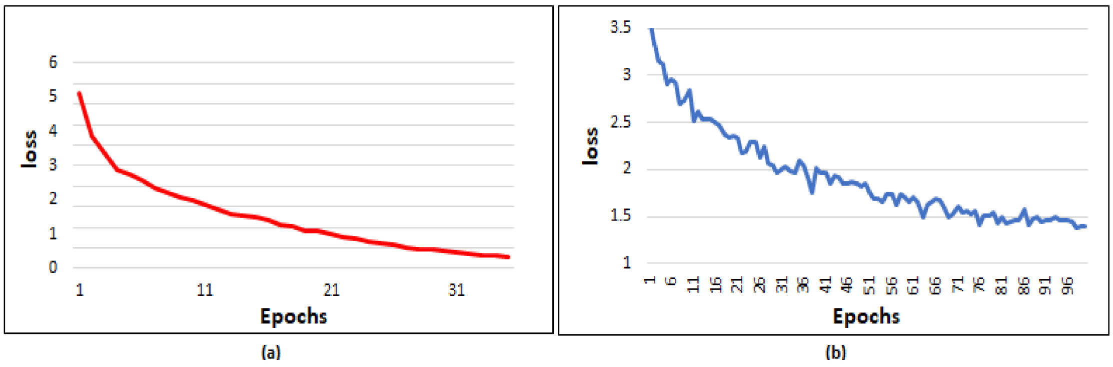


Figure 9. Transformer loss in training phase for (**a**) Django dataset and (**b**) SPoC dataset.

4.4. Results Discussion and Interpretation

After studying the models that are compared with the proposed model, we find the RNN model has two limitations. In the training step, the sentences are processed word by word and each word needs one training step, so the complexity of training is related to the length of the sentence, so the training of such model consumes the processing time. Second, the vanishing gradient problem of weights because information moves through time in RNNs associated with the previous step that caused a large change in weights and the weights will have a low value. Using the LSTM solves the vanishing gradient problem, but the model training is still word by word. The attention layer with RNN improves the result because the attention layer captures the long-term dependencies in input and output. However, these models may not capture the long-term dependencies between the code's tokens when the input sentence is long. In addition, failing to translate when the input tokens do not exist in the training dataset.

The retrieval mechanism based on NMT [15] has proven a high ability to manage either low-frequency tokens or do not exist tokens in the training dataset. Therefore, the proposed retrieval-based Transformer model adopts the retrieval methods with the transformer encoder-decoder architecture to achieve better performance.

The proposed retrieval-based Transformer model achieves better performance than other models because it processes the whole sentence in the training step. It uses the multi-head attention mechanism to make the alignment sentence better than RNN. With the observation, the results show that there is a limitation in handling sentences that do not exist in the training dataset or sentences containing tokens that do not exist in the training dataset. As a result of this problem, sentences that are not related to the input are generated and this distorts the prediction process, or the sentence is generated and does not contain the required tokens. This causes the tokens to be replaced with other tokens in the training dataset that are similar to them. Cross-validation solves this problem, but in reality, the input to the model can be a sentence that does not exist in the dataset. In this case, the same problem will occur even with the use of cross-validation. Therefore, the proposed model translates the sentence based on similar sentences in the dataset so that the translation output is accurate and then tokens are replaced with what matches the entered sentence. The limitation of the proposed model is that it may take a long time to retrieve entries of large datasets that consist of long sentences of more than 100 tokens. We plan to solve this problem by designing a learning model to improve the retrieval process as a question-answer model. We believe that this step allows the input data retrieval to take a constant time even with different dataset sizes.

5. Conclusions and Future Work

In this paper, a novel retrieval-based Transformer model is proposed for automatic pseudocode generation from the source code. The proposed model adapts many similarity methods to find the similar sentences of input is called the retrieved input. Then, it uses the encode-decoder transformer to translate retrieved input and the output is called the retrieved output. Finally, a replacement process is applied. This process starts by obtaining the word alignments and replacing tokens of the output retrieved in proportion to the tokens found in the input and input retrieved. The importance of the proposed model lies in dealing with the inputs that do not exist in the training dataset. The proposed model is evaluated using two datasets: Django and SPoC. The experimental results are promising compared to other methods. Using six and eight layers, it reaches 61.96 and 50.28 in terms of BLEU score performance measures for Django and SPoC datasets, respectively.

We are planning to add more languages such as C# and Java. Moreover, complicated code lines will be added to the source code. In addition, a pre-processing step is added to discover syntax errors to guarantee that the source code is correct before generating the pseudocode to increase retrieval-based transformer model performance.

Author Contributions: Conceptualization, W.G.; methodology, A.A. and W.G.; software, A.A. and W.G.; investigation, A.A., W.G. and W.N.; data curation, A.A.; writing—original draft preparation, A.A., W.G. and W.N.; writing—review and editing, W.G. and W.N.; Supervision on main idea, references, figures and experimental outputs, M.A. and A.-B.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Datasets used in experiments are public and they were retrieved from the following URLs: https://ahcweb01.naist.jp/pseudogen/ and https://sumith1896.github.io/spoc/ (accessed on 24 December 2021).

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

References

1. Gad, W.; Alokla, A.; Nazih, W.; Aref, M.; Salem, A.B. DLBT: Deep Learning-Based Transformer to Generate Pseudo-Code from Source Code. *Cmc-Comput. Mater. Contin.* **2022**, *70*, 3117–3132. [CrossRef]
2. Yang, G.; Zhou, Y.; Chen, X.; Yu, C. Fine-grained Pseudo-code Generation Method via Code Feature Extraction and Transformer. In Proceedings of the The 28th Asia-Pacific Software Engineering Conference (APSEC), Taipei, Taiwan, 6–9 December 2021.
3. Alhefdhi, A.; Dam, H.K.; Hata, H.; Ghose, A. Generating pseudo-code from source code using deep learning. In Proceedings of the The 25th Australasian Software Engineering Conference (ASWEC), Adelaide, SA, Australia, 26–30 November 2018; pp. 21–25.
4. Oda, Y.; Fudaba, H.; Neubig, G.; Hata, H.; Sakti, S.; Toda, T.; Nakamura, S. Learning to generate pseudo-code from source code using statistical machine translation. In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 9–13 November 2015; pp. 574–584.
5. Ling, W.; Blunsom, P.; Grefenstette, E.; Hermann, K.M.; Kocisky, T.; Wang, F.; Senior, A. Latent Predictor Networks for Code Generation. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, Berlin, Germany, 7–12 August 2016; Volume 1, pp. 599–609.
6. Jia, R.; Liang, P. Data Recombination for Neural Semantic Parsing. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, Berlin, Germany, 7–12 August 2016; Volume 1, pp. 12–22.
7. Locascio, N.; Narasimhan, K.; DeLeon, E.; Kushman, N.; Barzilay, R. Neural Generation of Regular Expressions from Natural Language with Minimal Domain Knowledge. In Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, Austin, TX, USA, 1–5 November 2016; pp. 1918–1923.
8. Babhulgaonkar, A.; Bharad, S. Statistical machine translation. In Proceedings of the 1st International Conference on Intelligent Systems and Information Management (ICISIM); IEEE: Manhattan, NY, USA; pp. 62–67.
9. Koehn, P. *Neural Machine Translation*; Cambridge University Press: Cambridge, UK, 2020.
10. Sennrich, R.; Zhang, B. Revisiting Low-Resource Neural Machine Translation: A Case Study. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, Florence, Italy, 28 July–2 August 2019; pp. 211–221.
11. Mahata, S.K.; Mandal, S.; Das, D.; Bandyopadhyay, S. Smt vs. nmt: A comparison over hindi & bengali simple sentences. *arXiv* **2018**, arXiv:1812.04898.
12. Yin, P.; Neubig, G. A Syntactic Neural Model for General-Purpose Code Generation. In Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, Vancouver, BC, Canada, 30 July–4 August 2017; Volume 1, pp. 440–450.
13. Rabinovich, M.; Stern, M.; Klein, D. Abstract Syntax Networks for Code Generation and Semantic Parsing. In Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, Vancouver, BC, Canada, 30 July–4 August 2017; Volume 1, pp. 1139–1149.

14. Roodschild, M.; Sardiñas, J.G.; Will, A. A new approach for the vanishing gradient problem on sigmoid activation. *Prog. Artif. Intell.* **2020**, *9*, 351–360. [CrossRef]
15. Zhang, J.; Utiyama, M.; Sumita, E.; Neubig, G.; Nakamura, S. Guiding Neural Machine Translation with Retrieved Translation Pieces. In Proceedings of the NAACL-HLT, Shenzhen, China, 18–22 July 2018; pp. 1325–1335.
16. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. *Adv. Neural Inf. Processing Syst.* **2017**, *30*, 5998–6008.
17. Büch, L.; Andrzejak, A. Learning-based recursive aggregation of abstract syntax trees for code clone detection. In Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 24–27 February 2019; pp. 95–104.
18. Deng, Y.; Huang, H.; Chen, X.; Liu, Z.; Wu, S.; Xuan, J.; Li, Z. *From Code to Natural Language: Type-Aware Sketch-Based Seq2Seq Learning. International Conference on Database Systems for Advanced Applications*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 352–368.
19. Gu, J.; Lu, Z.; Li, H.; Li, V.O. Incorporating Copying Mechanism in Sequence-to-Sequence Learning. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, Berlin, Germany, 7–12 August 2016; Volume 1, pp. 1631–1640.
20. Hayati, S.A.; Olivier, R.; Avvaru, P.; Yin, P.; Tomasic, A.; Neubig, G. Retrieval-Based Neural Code Generation. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, 31 October–4 November 2018; pp. 925–930.
21. Mou, L.; Li, G.; Zhang, L.; Wang, T.; Jin, Z. Convolutional neural networks over tree structures for programming language processing. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, Phoenix, AZ, USA, 12–17 February 2016.
22. Hu, X.; Li, G.; Xia, X.; Lo, D.; Jin, Z. Deep code comment generation. In Proceedings of the 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC), Gothenburg, Sweden, 27 May–3 June 2018; pp. 200–210.
23. Hu, X.; Li, G.; Xia, X.; Lo, D.; Jin, Z. Deep code comment generation with hybrid lexical and syntactical information. *Empir. Softw. Eng.* **2020**, *25*, 2179–2217. [CrossRef]
24. Rai, S.; Gupta, A. Generation of Pseudo Code from the Python Source Code using Rule-Based Machine Translation. *arXiv* **2019**, arXiv:1906.06117.
25. Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Liu, X. Retrieval-based neural source code summarization. In Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), Seoul, Korea, 5–11 October 2020; pp. 1385–1397.
26. Dumais, S.T. Latent semantic analysis. *Annu. Rev. Inf. Sci. Technol.* **2004**, *38*, 188–230. [CrossRef]
27. Salton, G.; Wong, A.; Yang, C.S. A vector space model for automatic indexing. *Commun. ACM* **1975**, *18*, 613–620. [CrossRef]
28. Liu, Z.; Xia, X.; Hassan, A.E.; Lo, D.; Xing, Z.; Wang, X. Neural-machine-translation-based commit message generation: How far are we? In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 3 September 2018; pp. 373–384.
29. Levenshtein, V.I. Binary codes capable of correcting deletions, insertions, and reversals. In Soviet Physics Doklady; Soviet Union. 1966. Volume 10. pp. 707–710. Available online: https://nymity.ch/sybilhunting/pdf/Levenshtein1966a.pdf (accessed on 24 December 2021).
30. Reiter, E. A structured review of the validity of BLEU. *Comput. Linguist.* **2018**, *44*, 393–401. [CrossRef]
31. Haiduc, S.; Aponte, J.; Moreno, L.; Marcus, A. On the use of automated text summarization techniques for summarizing source code. In Proceedings of the 17th Working Conference on Reverse Engineering, Beverly, MA, USA, 13–16 October 2010; pp. 35–44.
32. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. Pytorch: An imperative style, high-performance deep learning library. *Adv. Neural Inf. Processing Syst.* **2019**, *32*, 8026–8037.
33. Honnibal, M.; Montani, I.; Van Landeghem, S.; Boyd, A. *spaCy: Industrial-Strength Natural Language Processing in Python*; Zenodo: Geneva, Switzerland, 2020.
34. Princeton University. About WordNet. Available online: https://wordnet.princeton.edu/ (accessed on 3 March 2021).