Computer Modeling in Engineering & Sciences

Tech Science Press

DOI: 10.32604/cmes.2023.024018

check for updates

ARTICLE

Implementation of Rapid Code Transformation Process Using Deep Learning Approaches

Bao Rong Chang[1], **Hsiu-Fen Tsai**[2,*] **and Han-Lin Chou**[1]

[1]Department of Computer Science and Information Engineering, National University of Kaohsiung, Kaohsiung, 811, Taiwan

[2]Department of Fragrance and Cosmetic Science, Kaohsiung Medical University, Kaohsiung, 811, Taiwan

*Corresponding Author: Hsiu-Fen Tsai. Email: sftsai@kmu.edu.tw

Received: 21 May 2022 Accepted: 05 September 2022

ABSTRACT

Our previous work has introduced the newly generated program using the code transformation model GPT-2, verifying the generated programming codes through simhash (SH) and longest common subsequence (LCS) algorithms. However, the entire code transformation process has encountered a time-consuming problem. Therefore, the objective of this study is to speed up the code transformation process significantly. This paper has proposed deep learning approaches for modifying SH using a variational simhash (VSH) algorithm and replacing LCS with a piecewise longest common subsequence (PLCS) algorithm to faster the verification process in the test phase. Besides the code transformation model GPT-2, this study has also introduced Microsoft MASS and Facebook BART for a comparative analysis of their performance. Meanwhile, the explainable AI technique using local interpretable model-agnostic explanations (LIME) can also interpret the decision-making of AI models. The experimental results show that VSH can reduce the number of qualified programs by 22.11%, and PLCS can reduce the execution time of selected pocket programs by 32.39%. As a result, the proposed approaches can significantly speed up the entire code transformation process by 1.38 times on average compared with our previous work.

KEYWORDS

Code transformation model; variational simhash; piecewise longest common subsequence; explainable AI; LIME

1 Introduction

Artificial intelligence (AI) [1] has returned to mainstream technology and developed deep learning in recent years. With the development of deep learning, the Google Brain team has developed Tensorflow, which can use for machine learning of various perception and language understanding tasks and data prediction, image feature classification, and text generation. Deep learning has introduced generation-based models, including long short-term memory (LSTM) [2] and generative adversarial network (GAN) [3]. Although LSTM can generate text, it only predicts the words that may appear next and cannot generate similar articles. In contrast, GAN generates pictures better than text. To allow machines to imitate humans more accurately, the trend of AI applications is gradually developing toward human language-related issues.

As the technology of AI becomes more and more mature, the related natural language processing has also developed rapidly shortly. In terms of language processing, the English natural language sentence segmentation model, natural language toolkit (NLTK) [4] based on Python programming, has been developed for years. A well-trained NLTK can segment English text into sentences or words and perform text processing such as part-of-speech tags. For more advanced text conversion models, the non-profit organization OpenAI LP has developed GPT-2 [5], the second generation of generative pre-training tool and belongs to the unsupervised learning transformer model. It is mainly engaged in English imitation creation in artificial intelligence. Besides text, people can also use it to generate fake news [6]. In addition to GPT-2, Microsoft has developed MASS and Facebook AI BART. They can also do a similar job like GPT-2 very well. Is it possible to think of a similar manner to use them to generate the code of a designated programming language, for example, the code of Python programming? The previous paper [7] introduced the newly generated program using the code transformation model GPT-2, where users can verify the generated programming codes through simhash (SH) and longest common subsequence (LCS) algorithms. However, the code transformation process has encountered the problem of being time-consuming in the previous work.

Therefore, the objective of this study is to propose the deep learning approaches for modifying SH and LCS algorithms where a modified SH (called variational simhash) can reassign the appropriate weighted value of keywords, and a modified LCS (called piecewise longest common subsequence) can break a long string into a few smaller pieces of string. In such a way, the proposed approaches can significantly reduce the algorithm execution time to speed up the entire process of code transformation, which is our main contribution to this study. Meanwhile, this study also introduces three code transformation models, GPT-2, MASS, and BART, and will deliver the performance evaluation among them and check which one can achieve better results.

The following paragraphs of this paper are arranged as follows. In Session 2, related work will be described in text transform models, content check, predetermined generative programs, and explainable AI techniques. The way to system implementation is given in Session 3. The experimental results and discussion will be obtained in Session 4. Finally, we drew a brief conclusion in Session 5.

2 Related Work

2.1 Text Transform Model

The traditional text transformer model contains an Encoder and Decoder stack, as shown in Fig. 1. GPT-2 [5] is an unsupervised transformer language generation model released by OpenAI in 2019. GPT-2 is composed of a Decoder stack, as shown in Fig. 2. This study has received much programming information related to transformer-based models like GPT-2 from the website Github for training GPT-2 as a code transformation model.

MASS [8] is a transformer-based model that connects the Encoder and Decoder by attention. A continuous segment of length k randomly shields the input of the Encoder of MASS compared to the basic transformer architecture. After that, it uses a Decoder to predict the masked segment, as shown in Fig. 3. BART [9] is intrinsically the architecture of a standard transformer model. Like MASS, it masks the Encoder's input, but the difference is that the masked words can be a few discontinuous segments, as shown in Fig. 4. BART replaces the excitation function ReLU with GeLUs, and Decoder has not changed. The loss function of BART is the cross-entropy between the output of the Decoder and the original text. BART uses this cross-entropy loss to optimize its transform model and improve accuracy.

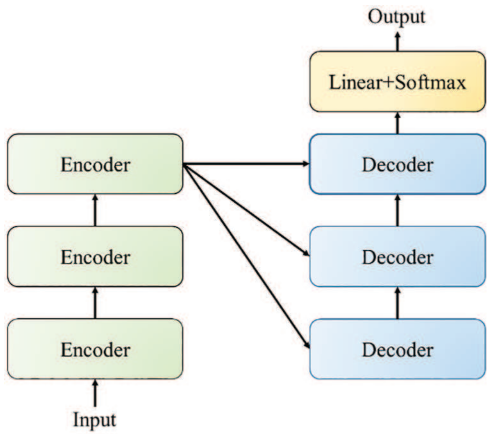


Figure 1: Traditional text transformer model

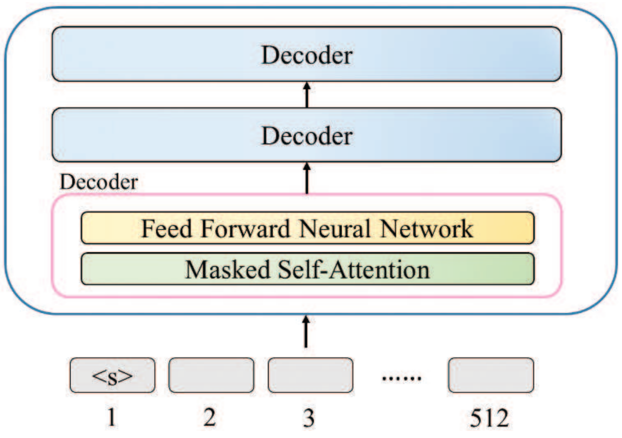


Figure 2: GPT-2 architecture

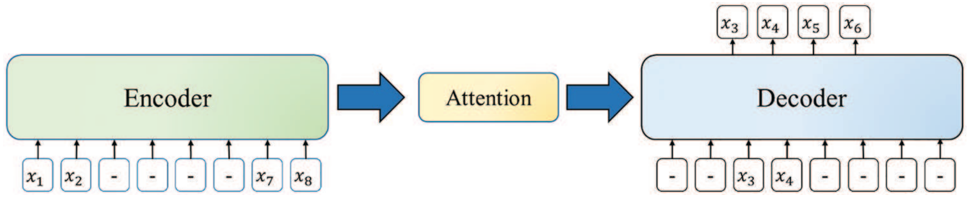


Figure 3: MASS architecture

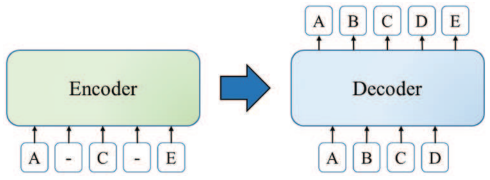


Figure 4: BART architecture

2.2 *Binary Code Comparison*

Simhash [10] is a kind of locality-sensitive hash, and its main idea is to reduce dimensionality, as shown in Fig. 5. It maps the high-dimensional feature vector to the low-dimensional feature vector and compares the two vectors with the similarity of the article through the Hamming distance [11]. The smaller the Hamming distance, the higher the similarity.

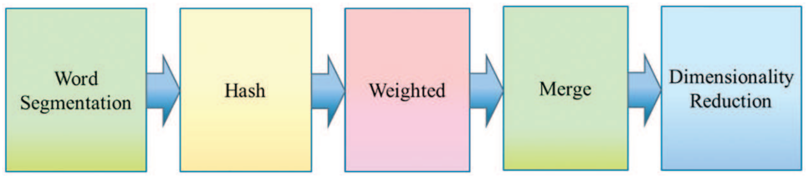


Figure 5: SimHash process flow

After the text is segmented, the Hash calculation obtains the feature vector. Adding weight to the feature vector is weighting and accumulating all the weighted vectors. The cumulative result is more significant than zero as one and less than zero as zero, and users can obtain the fingerprint, as shown in Fig. 6. Finally, we calculate the two fingerprints [12] for Hamming distance.

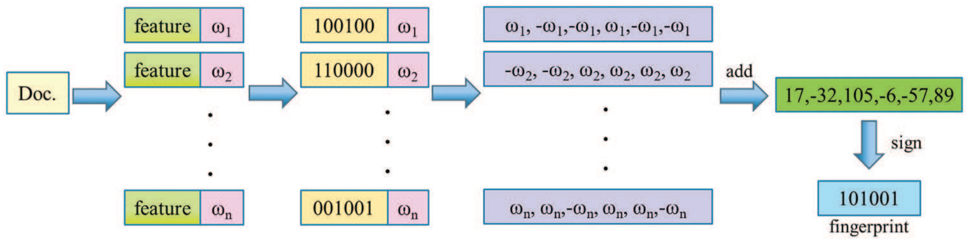


Figure 6: Calculate fingerprint

Generally speaking, the simhash algorithm uses the formula of tf-idf to estimate the corresponding weight value for each keyword. In Eq. (1), idf_i is referred to as inverse document frequency, which refers to a measure of the universal importance of a word. Furthermore, f_i represents the number of files in which the word i appears in the text j, and D is the total number of all texts. In Eq. (2), $tf_{i,j}$ represents term frequency meaning the average frequency of the word i in the text j, and $f_{i,j}$ stands for the frequency of the word i appearing in the text j. $\sum_{i'\in j} i',j$ is the sum of the frequency of each word appearing in the text j, which is the length of the text. In Eq. (3), $tfidf_{i,j}$ is the estimated weight value of the specific word, $tf_{i,j}$ is the term frequency meaning the frequency of a word given in the file, and idf_i is the inverse document frequency, a measure of the universal importance of a word given in the file.

$$idf_i = \log\frac{D}{f_i}, f_i = |\{j \in D, i \in j\}| \tag{1}$$

$$tf_{i,j} = \frac{f_{i,j}}{\sum_{i'\in j} i',j} \tag{2}$$

$$tfidf_{i,j} = tf_{i,j} * idf_i \tag{3}$$

2.3 Multimedia Content Check

Although there are many methods to compare different forms of media information, this study uses LCS [13] as a single effective method to detect the consistency of multimedia information, which is suitable for text, pictures, sounds, and movies. LCS finds the longest common subsequence in the sequence set. Unlike searching for the longest common substring, the position of the subsequence in the original sequence does not need to be continuous, as shown in Fig. 7. The yellow part in Fig. 7 indicates the completion of the comparison. In this study, we convert the execution result of each qualified program and the execution result of the corresponding sample program into ASCII code or binary code and use LCS to check the conformity. To check whether the execution result of the generated program is consistent with the execution result of the sample program, please confirm that the generated program is available.

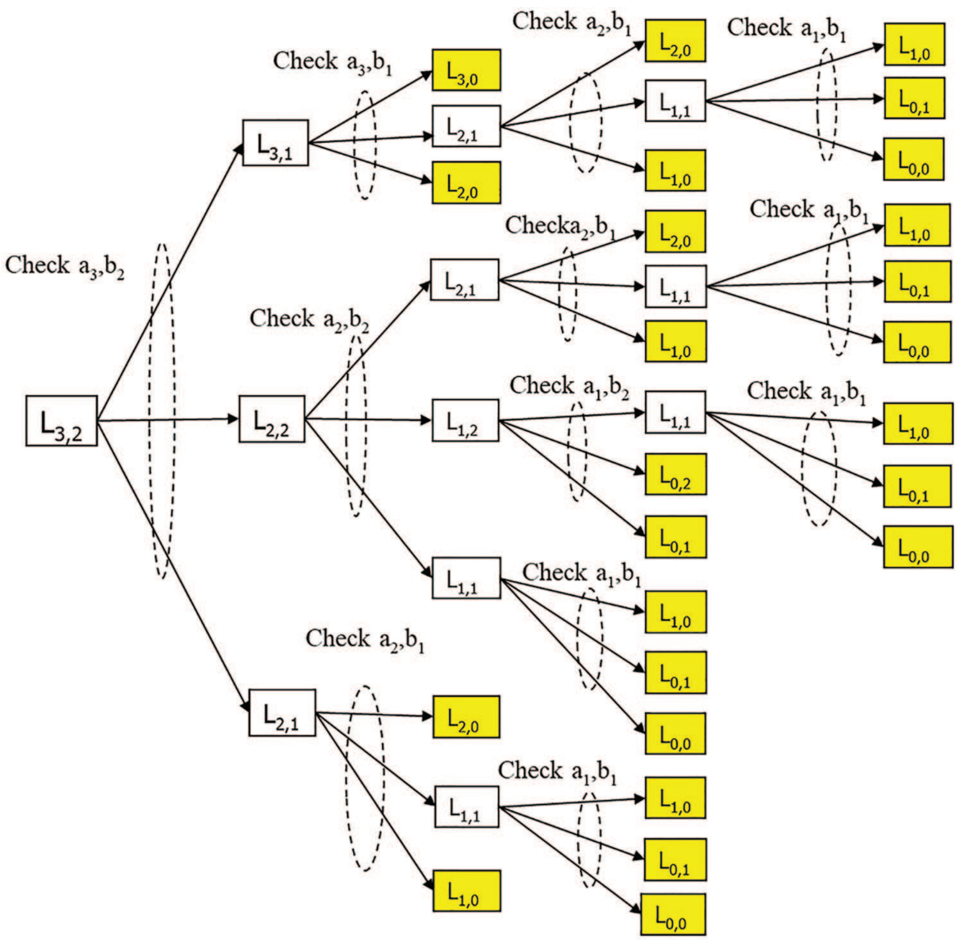


Figure 7: Longest common subsequence (LCS)

2.4 Predetermined Generative Programs

We do not know the number of the code-transformed programs produced from GPT-2, MASS, and BART is appropriate and needed. Therefore, this study theoretically introduces a statistical estimation of the number of predetermined code-transformed programs, and hereafter the preliminary programs denote the code-transformed programs. This estimation implies how many preliminary programs are needed to ensure that the code transformation process can find the best-performance one. Therefore, users can first calculate the quantity of the preliminary programs generated by a sample program with a pass ratio exceeding 90% and obtain the ratio q_i, as shown in Eq. (4). In Eq. (4), N_{si} represents the total number of primary programs, and x_{si} stands for the number of primary programs whose pass ratio exceeds 90%. After the previous calculation, Eq. (5) gets the average ratio q, where t is the total number of sample programs. Then users can find out the number of misjudgments in, x_{si} and calculate the ratio of misjudgments q_{mi}, as shown in Eq. (6). In Eq. (6), x_{msi} represents the number of misjudgments among the number of the preliminary programs with a pass ratio of more than 90%. Then Eq. (7) can obtain the average percentage of misjudgments q_m.

$$q_i = \frac{x_{si}}{N_{si}}, i = 1, 2, \ldots, t \tag{4}$$

$$q = \frac{\sum_{i=1}^{t} q_i}{t} \tag{5}$$

$$q_{mi} = \frac{x_{msi}}{x_{si}}, i = 1, 2, \ldots, t \tag{6}$$

$$q_m = \frac{\sum_{i=1}^{t} q_{mi}}{t} \tag{7}$$

After that, users can calculate the number of the preliminary programs whose pass ratio is less than 90% to get the ratio u_i, as shown in Eq. (8). In Eq. (8), y_{si} represents the number of the preliminary programs whose pass ratio is less than 90%. Next, Eq. (9) finds out the average ratio u. Then users can find out the number of misjudgments y_{si} and then calculate the proportion of misjudgments u_{mi}, as shown in Eq. (10). In Eq. (10), y_{msi} represents the number of misjudgments. The preliminary programs have many misjudgments with less than a 90% pass ratio. Finally, Eq. (11) gives the average percentage of misjudgments u_m.

$$u_i = \frac{y_{si}}{N_{si}}, i = 1, 2, \ldots, t \tag{8}$$

$$u = \frac{\sum_{i=1}^{t} u_i}{t} \tag{9}$$

$$u_{mi} = \frac{y_{msi}}{y_{si}}, i = 1, 2, \ldots, t \tag{10}$$

$$u_m = \frac{\sum_{i=1}^{t} u_{mi}}{t} \tag{11}$$

Eq. (12) counts the total of preliminary programs generated by all the sample programs to get N_g. According to the statistics such as N_g, q, q_m, u, and u_m, Eq. (13) obtains the average pass ratio of over a 90% probability P_{gq}. Assuming that a pass ratio of j programs exceeds 90%, $P(K_j|P_{gq})$ means that for these j programs, the probability that a pass ratio of the code similarity check exceeds 90% is valid, such as Eq. (14). In Eq. (14), K_j represents j programs with a pass ratio of code similarity check of 90% or more among the generated programs. According to the above statistics, the probability of the code similarity check with the pass ratio of j programs exceeding 90% is $P(K_j|P_{gq})$. Therefore, Eq. (15) can infer the minimum number of preliminary programs produced from the code transformation process to ensure that at least j programs have a code similarity check with a pass ratio of more than 90%, where N is the total number of preliminary programs to be generated.

$$N_g = \sum_{i=1}^{t} N_{si} \tag{12}$$

$$P_{gq} = \frac{N_g \cdot q \cdot (1 - q_m) + N_g \cdot u \cdot u_m}{N_g} \tag{13}$$

$$P(K_j|P_{gq}) = \frac{P(K_j \cap P_{gq})}{P_{gq}} \tag{14}$$

$$N \cdot P(K_j|P_{gq}) \geq K_j \tag{15}$$

Take four sample programs of the previous paper [7] as an example. Each sample program can generate 500 preliminary programs, respectively, and Table 1 gives the average percentage of the preliminary programs with a pass ratio over or less than 90%. We can calculate the probability P_{gq} indicating the preliminary programs with a pass ratio of code similarity check over indeed 90% is 8.6% in Eq. (13). Next, Eq. (14) can estimate the probability $P(K_j|P_{gq})$ representing the preliminary programs with a pass ratio of code similarity check over 90% is 4.91%. Suppose there are 5 preliminary programs with a pass ratio of code similarity check over 90% required. In that case, Eq. (15) can infer that a specific code transformation model should produce at least 100 preliminary programs for use.

Table 1: The average percentage of preliminary programs with a pass ratio over or less than 90%

	Program group	
Average percentage	Preliminary programs with a pass ratio of code similarity check over 90% where $q = 10\%$	Preliminary programs without a pass ratio of code similarity check over 90% where $u = 90\%$
The average percentage of no false positives	81.78%	92.33%
The average percentage of false positives	18.22%	7.67%
The probability of a pass ratio of code similarity check over 90%	8.6%	

2.5 *Explainable AI Technique*

Most data scientists prefer high-precision metrics when using models to solve problems, and high-precision models are too complex to understand the decision-making of their algorithms. Because the algorithms are too complex, the designers can consistently not explain why artificial intelligence can achieve specific effects. Today, researchers propose explainable artificial intelligence so that people can understand the inference of the results from AI-related algorithms. There are three classification criteria for interpretability methods: (1) Essential or ex-post interpretability, (2) Model-specific or general, and (3) Local or global interpretability. In 2016, Marco Ribeiro, Sameer Singh, and Carlos Guestrin proposed local interpretable model-agnostic explanations (LIME) [14], a post-analytical approach to interpretation after model establishment, as shown in Fig. 8. The first is randomly selecting attributes from the data as data perturbations and other labels the results. Next, in LIME, we calculate the distance between the result and the attributes to get the weights of the attributes and then filter the attributes with small weights. After that, we can explain the result according to the remaining attributes. Therefore, people can use explainable AI techniques to supervise the rules discovered by AI systems, and judging those rules can explain the outcomes of their decisions.

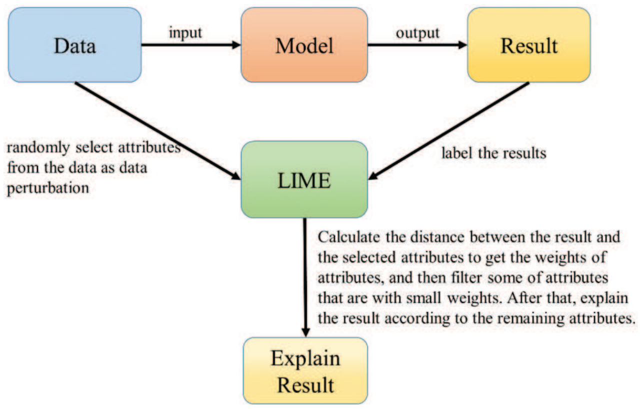

Data
input
Model
output
Result
randomly select attributes from the data as data perturbation
label the results
LIME
Calculate the distance between the result and the selected attributes to get the weights of attributes, and then filter some of attributes that are with small weights. After that, explain the result according to the remaining attributes.
Explain Result

Figure 8: LIME processing flow

3 Research Method

The previous paper [7] used a code transformation model to generate similar and more concise programs that can shorten execution times. Users can retrieve the corresponding sample programs from the semantic database using keywords. However, the retrieved sample programs may encounter the problem of their codes in low execution performance. Therefore, the previous work is to find a way to transform its sample program into the newly generated programs that can perform better than the original one magnificently because we have significantly improved the execution speed of the entire code transformation process in this study. The previous paper introduced the code transformation model using GPT-2 to produce the newly generated programs called the preliminary programs. Some of the preliminary programs are not exploitable. Therefore, we must confirm which programs are exploitable through the code similarity check and the execution output conformity check. Then the system chooses the best-performing one as a pocket program for this instance, as shown in Fig. 9.

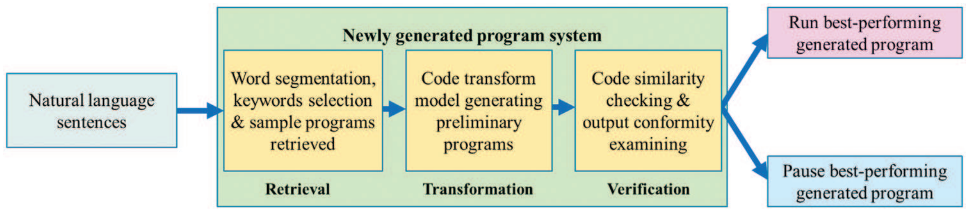

Newly generated program system
Natural language sentences
Word segmentation, keywords selection & sample programs retrieved
Code transform model generating preliminary programs
Code similarity checking & output conformity examining
Retrieval
Transformation
Verification
Run best-performing generated program
Pause best-performing generated program

Figure 9: Natural language generating program process

3.1 Code Transformation Process

The code transformation process is done by inputting the sentence, selecting keywords after NLTK segmentation, searching the sample programs by keywords, and feeding the sample programs into the code transformation model producing the newly generated programs with better efficiency to improve their execution speed. The entire procedure includes two stages, model generation, and model use stages, as shown in Figs. 10 and 11. The first is the training phase in the model generation stage, and the next is the test phase. The training phase contains several units: word segmentation, sample program, generative program model, and generated program units. After the word segmentation, the system uses keywords to search sample programs from the semantic database to find the proper sample programs corresponding to the action initiated by the input sentence. To train the program generation model, users can submit the retrieved sample programs to the pre-trained code transformation model, e.g., GPT-2. Once users have established the program generation model, they can feed it into that code transformation model again to produce the newly generated programs. They are called preliminary programs. The first step uses the variational simhash algorithm to check code similarity between the sample program and anyone of the preliminary programs in the test phase. Then the next step uses PLCS to check the conformity of the execution results between sample programs and anyone of qualified programs. Two steps are to filter the un-qualified programs and leave a few programs with higher consistency in code similarity and output conformity. Finally, users can pick out the best-performing program as a pocket program and save it into the semantic database to replace the original sample program. Since input sentences and the program's code are composed of words and symbols, they are all pure text. Therefore, the training and test phases can successfully handle text streams to achieve a unified parallel analysis.

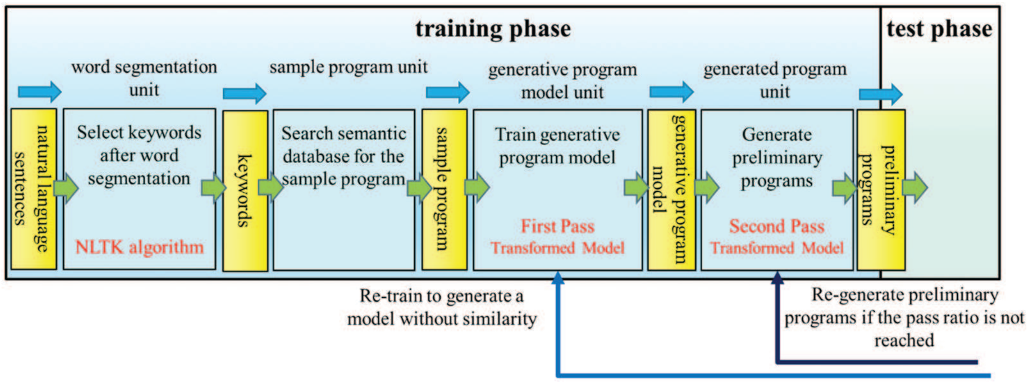

training phase
test phase
word segmentation unit
sample program unit
generative program model unit
generated program unit
natural language sentences
Select keywords after word segmentation
NLTK algorithm
keywords
Search semantic database for the sample program
sample program
Train generative program model
First Pass Transformed Model
generative program model
Generate preliminary programs
Second Pass Transformed Model
preliminary programs
Re-train to generate a model without similarity
Re-generate preliminary programs if the pass ratio is not reached

Figure 10: Training phase of model generation stage

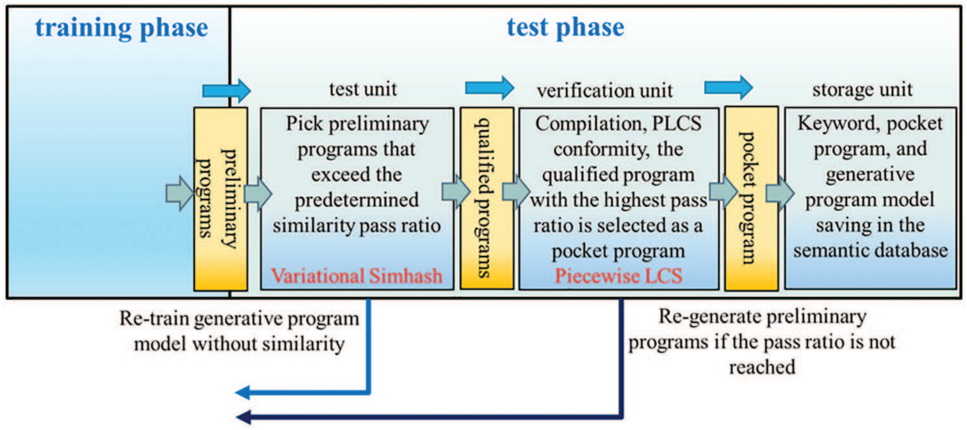

training phase
test phase
test unit
verification unit
storage unit
preliminary programs
Pick preliminary programs that exceed the predetermined similarity pass ratio
Variational Simhash
qualified programs
Compilation, PLCS conformity, the qualified program with the highest pass ratio is selected as a pocket program
Piecewise LCS
pocket program
Keyword, pocket program, and generative program model saving in the semantic database
Re-train generative program model without similarity
Re-generate preliminary programs if the pass ratio is not reached

Figure 11: Test phase of model generation stage

In the model use stage, as shown in Fig. 12, if users can retrieve the pocket program from the semantic database directly, they go to execute it to retain the output. Otherwise, the code transformation process is like the model generation stage until the step has found the pocket program. After that, users can run the pocket program to obtain the execution result at once. Similarly, the model use stage can successfully deal with text streams between tasks, as mentioned earlier in the model generation stage.

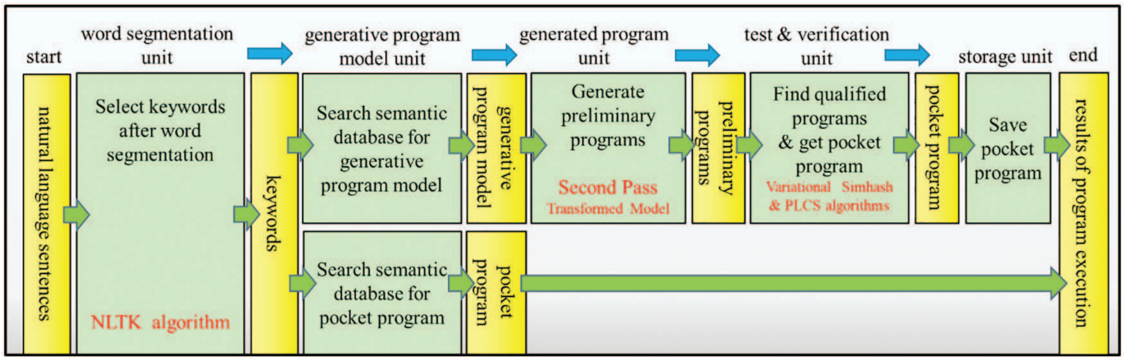

start
word segmentation unit
generative program model unit
generated program unit
test & verification unit
storage unit
end
natural language sentences
Select keywords after word segmentation
NLTK algorithm
keywords
Search semantic database for generative program model
Search semantic database for pocket program
generative program model
pocket program
Generate preliminary programs
Second Pass Transformed Model
preliminary programs
Find qualified programs & get pocket program
Variational Simhash & PLCS algorithms
pocket program
Save pocket program
results of program execution

Figure 12: Model use stage

3.2 Variational Simhash Algorithm (VSH)

Generally speaking, the weighting method of simhash uses the TF-IDF algorithm as its weighting value. Its method counts the frequency of word occurrence, which is suitable for text comparison. But in code, unreserved words appear frequently but are not very important. Suppose we obtain the weights according to the original method. In that case, the weight of the non-reserved words will be greater than the weight of the reserved words, making the code similarity inaccurate, so the original method is unsuitable for the code. Therefore, this study proposed a variational autoencoder model suitable for code similarity comparison, called the variational simhash algorithm, and Fig. 13 gives its algorithm. This study will first train a set of variational autoencoder (VAE) to give weights that are reserved words greater than symbols and symbols greater than non-reserved words. After that, convert all reserved words, symbols, and non-reserved words in the code to be compared to an n-bit vector via word2vec. The weights are then directly mapped to an m-bit vector via the VAE. Compared with user-defined weights used in the simhash algorithm, the proposed variational simhash algorithm can provide weights much closer to the normal distribution. Since the user-defined weights don't follow any protocol or regulation, the traditional simhash algorithm cannot provide the appropriate weight for each corresponding word. Suppose a word does not exist in the list of defined words. The traditional algorithm cannot give it appropriate weight. On the contrary, the proposed approach can assign a proper weight based on the VAE's inference.

This study uses VAE to map weights directly. First, convert a word into a 9-dimensional vector through word2vec, as shown in Fig. 14. The input layer is this vector. The Encoder has two layers, the first layer has seven neurons, and the second layer has five neurons. We then reduce the vector to a 3-dimensional vector by the mean and standard deviation, and the Decoder is responsible for restoring the data dimension to the original dimension. Then go back to the word via word2vec. This study found 550 codes from GitHub, of which 500 are training data, 40 are validation data, and 10 are test data. The loss function is MSE, the activation function is ReLU, and the optimizer is Adam. Fig. 15 shows the architecture diagram.

```
Algorithm. Variational Simhash
input : Sample program, Generating program
output : Similarity
1.  function NLTK (Sample program, Generating program)
2.  return Sample program's keywords, Generating program's keywords
3.  function Variational AutoEncoder (Sample program's keywords, Generating program's keywords)
4.  return keyword's weight
5.  Num (Sample program) = hash (Sample Program's keywords) * keyword's weight
6.  Num (Generating program) = hash (Generating program 's keywords) * keyword's weight
7.  Result (Sample program) += Num (Sample Program)
8.  Result (Generating program) += Num (Generating program)
9.  Binary (Result (Sample Program))
10. Binary (Result (Generating program))
11. function hamming (Binary (Result (Sample Program)), Binary (Result (Generating program)))
12. return Similarity
```

Figure 13: Variational simhash algorithm

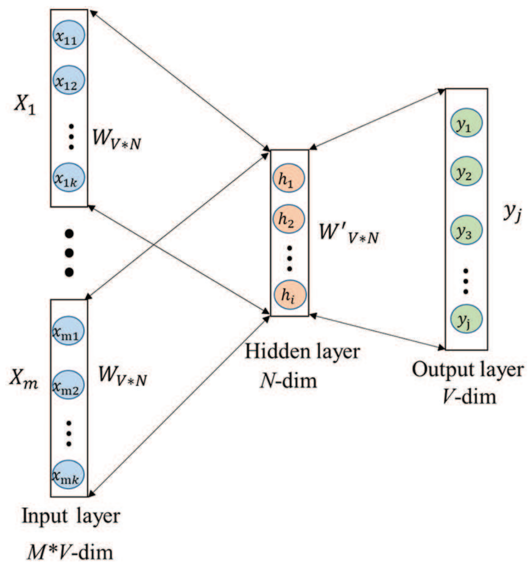


Figure 14: word2vec architecture

After reducing to a three-dimensional vector, VAE substitutes the data into Eq. (16) to obtain the weight value, where i is the number of neurons in the hidden layer, h_i represents the value of the i-th neuron, and W stands for the weight value. This study has carried out an example with two simple program codes in Test1 [15] and Test2 [16] to demonstrate the efficient implementation of two methods, word2vec and VAE. It turned out that the simhash algorithm gave 68% of the code similarity check. The variational simhash algorithm, by contrast, inferred it to be 90%.

$$W = \left(\sum_{i=1}^{n} h_i\right)^{\frac{1}{2}} \tag{16}$$

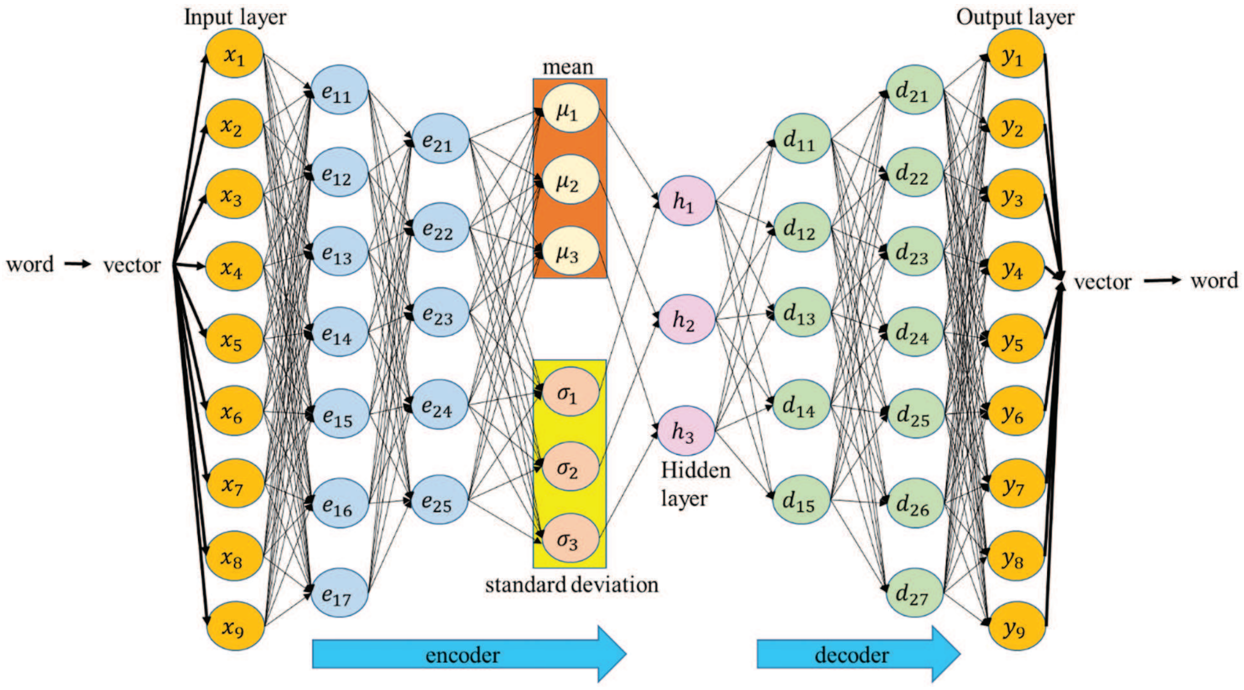


Figure 15: Variational autoencoder (VAE) architecture

3.3 Piecewise Longest Common Subsequence (PLCS)

This study proposed a new effective LCS-like method to rapidly check multimedia information's consistency. After converting the execution result of each program into a string of ASCII code or binary code, we use LCS to compute the conformity of the sample program and the generated program execution results. This study found that when the length of a string of ASCII or binary code is very long, it takes a long time to finish the conformity check. Technically speaking, supposed two strings with the length of n individually, LCS will spend n^2 times of comparisons to check the conformity between them. This study has proposed an improved LCS algorithm called the piecewise longest common subsequence (PLCS) to shorten the conformity check, as shown in Fig. 16. PLCS can use a deep neural network (DNN) to predict the appropriate segmented length of a string of ASCII or binary code. First, it converts the execution result into a string ASCII or binary code, breaking it into several segments where a segment has a fixed length. After that, it uses the LCS algorithm to perform a conformity check segment by segment. After the algorithm completes the LCS operation on each segment, it will empty the memory allocated for the calculation. The algorithm will return only the LCS result of the segment as well. Finally, we add the LCS results of the segments to get the final LCS result. Supposed the length of the two strings is n, the algorithm derives every segment with k characters to perform PLCS. The PLCS will spend $k^2 * \left\lfloor \frac{n}{k} \right\rfloor + (n\, mod\, k)^2$ times of the comparison. The number of comparisons used in the proposed approach, PLCS, is much less than the traditional method LCS. The operation of PLCS is faster than that of LCS because a small amount of memory is allocated for a single segment computing to speed up the conformity operation. In theory, the data type or the length of the string affects how long the segment length set in the string should be. Therefore, this study employs a DNN model to predict how many characters combine a segment to infer segment lengths for long sequences as the most suitable way to compute PLCS quickly.

```
Algorithm . Piecewise longest common subsequence – PLCS
input : Sample program's result, Generating program's result
output : timer
1.  Timer start
2.  function NLTK (Sample program's result, Generating program's result)
3.  return Sample program's keywords, Generating program's keywords
4.  Binary (Sample program's keywords)
5.  Binary (Generating program's keywords)
6.  Len (Binary (Sample program's keywords))
7.  Len (Binary (Generating program's keywords))
8.  function DNN (feature, signal length, signal width)
9.  return How much length to do cutting
10. array =cut (Binary (Sample program's keywords), Binary (Generating program's keywords))
11. function LCS (array)
12. return LCS
13. result += LCS
14. Timer end
15. Timer = Timer end – Timer start
```

Figure 16: PLCS algorithm

This study employs a deep neural network (DNN) model with a softmax function to predict the length of a segment, as shown in Fig. 17. In Fig. 17, the symbol from O_1 to O_5 represents the different lengths of a segment, and we specify these symbols to different lengths of a segment, as listed in Table 2. The input layer contains three parts: feature and the length and width of the signal. The loss function is the sum of squared errors (SSE), the activation function is the rectified linear unit (ReLU), and the optimizer is the adaptative gradient (AdaGrad). The feature input is a one-dimensional vector with a length of 784 elements. The data set for training a DNN model consists of 100 input vectors and their corresponding 100 output labels where the model performs segmentation results of different lengths.

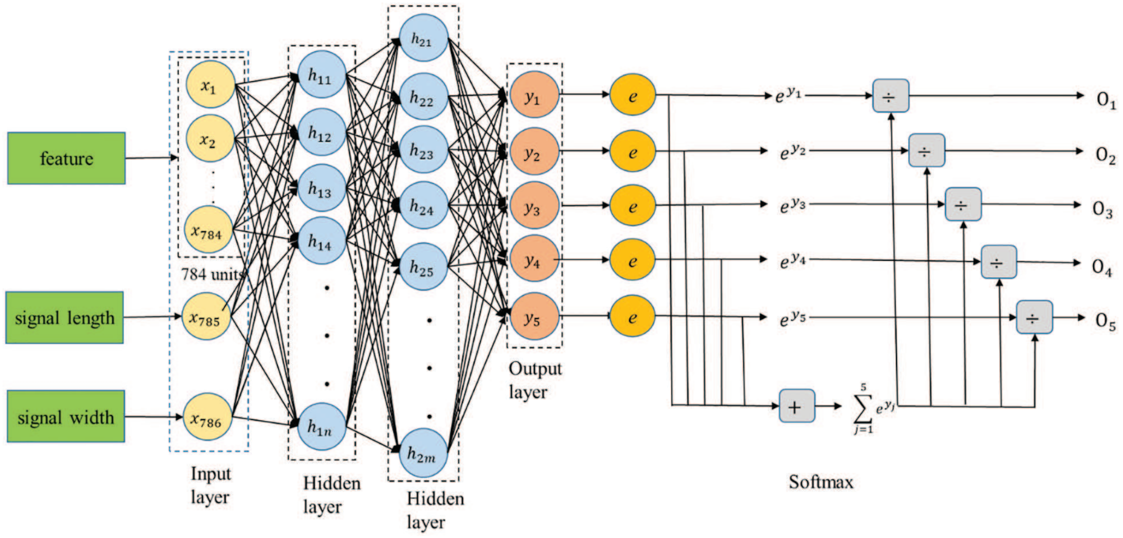

feature
signal length
signal width
x_1
x_2
x_{784}
784 units
x_{785}
x_{786}
Input layer
h_{11} h_{12} h_{13} h_{14} h_{1n}
Hidden layer
h_{21} h_{22} h_{23} h_{24} h_{25} h_{2m}
Hidden layer
y_1 y_2 y_3 y_4 y_5
Output layer
e
e^{y_1} e^{y_2} e^{y_3} e^{y_4} e^{y_5}
÷
+
$\sum_{j=1}^{5} e^{y_j}$
O_1 O_2 O_3 O_4 O_5
Softmax

Figure 17: DNN estimate the length of a segment

Table 2: The length of a segment (unit: bit)

Symbol	One hot encoding	Length of a segment
O_1	00001_2	500
O_2	00010_2	1000
O_3	00100_2	5000
O_4	01000_2	10000
O_5	10000_2	50000

Regarding data allocation for training a DNN model, there are 90 vectors as training data, five validation data, and five test data. There are two hidden layers in DNN. The number of neurons in the first hidden layer is 30 and in the second 40. If the input signal is text, the feature value of the input layer adopts Doc2Vec to convert the text into a vector, and the signal length is the original length of the sentence in which the signal width is 1. Users can use VGG16 to capture image features as an input signal if the input is an image. The signal length is the original length of the image, and the signal width is the image's original width. If the input is a voice signal, users can use the Python package librosa.display.waveplot to convert it into the waveform of an image as an input signal. The training process of a voice input will do the same task as the image input process mentioned above. If the input is a movie, users can use the YOLO v3 model to track the object's motion and convert the track of motion into a displacement image as an input signal. The training process of a move input will do the same task as the image input process mentioned above. Fig. 18 shows the loss curve during the training phase of a DNN model.

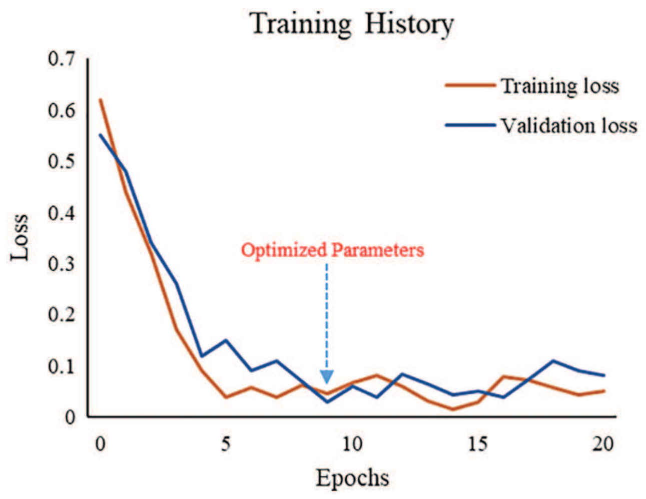


Figure 18: Loss curve during DNN training phase

This study uses the execution results of four sample programs and the programs they generate for testing. The execution results are article text [17], graphic image [18], speech signal [19] and video signal [20], respectively. Table 3 shows the performance evaluation between LCS and PLCS. The LCS algorithm takes 659.50 s on average for the conformity check of execution results. In contrast, the PLCS algorithm is 173.25 s.

Table 3: Performance evaluation between LCS and PLCS

Case	Predicted segment length (bit)	Number of comparisons using LCS	LCS execution time (s)	Number of comparisons using PLCS	PLCS execution time (s)
Article text	500	485,809	0.006	288,809	0.0017
Graphic image	1000	1,227,241,024	298	35,001,024	78
Voice signal	10000	21,278,640,384	1397	1,434,480,384	431
Video signal	5000	6,430,917,249	943	400,037,249	184
Average	4125	7,234,321,117	659.50	467,451,866.5	173.25

4 Experimental Results and Discussion

4.1 Experiment Setting

This study uses fast model training on an advanced GPU [21] cluster architecture to reduce the processing time spent on traditional CPU training models, as shown in Fig. 19. NLTK performed sentence segmentation and keyword searches to find the corresponding sample programs, and then users fed those programs into code transformation models GPT-2, MASS, and BART. The variational simhash algorithm checks for code similarity. The Piecewise Longest Common Subsequence algorithm checks the consistency of the execution results of two different programs. Use LIME to interpret the model. Users can build all of the above tools in a cloud environment to execute most applications and generate programs. Therefore, this study uses open source packages to establish the operating environment, as listed in Table 4.

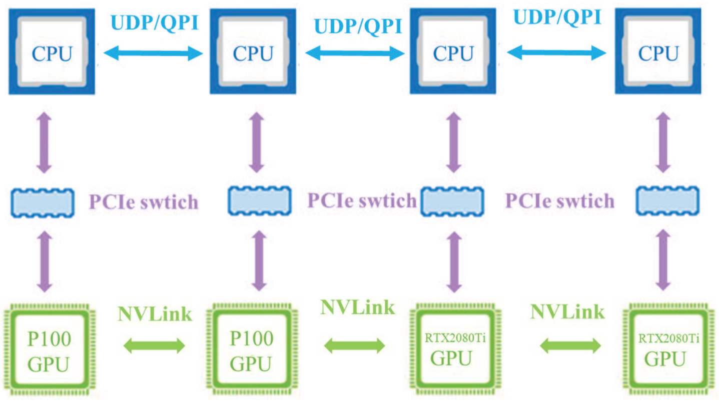


Figure 19: GPU workstation cluster

Table 4: Open-source package

Package	Version
Anaconda2	5.2.0
Python	3.7.5

(Continued)

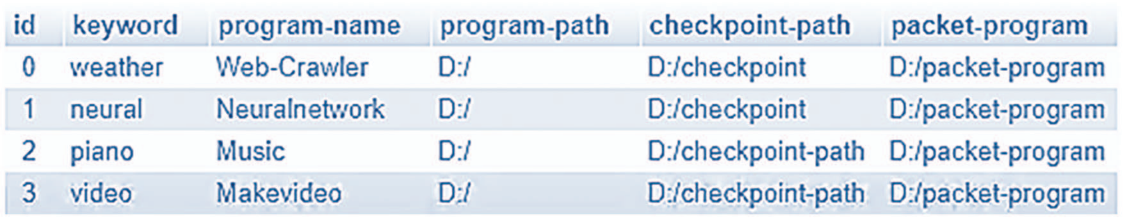
Table 4 (continued)

Package	Version
Tensorflow	1.14
CUDA	10
XAMPP	3.2.4
NLTK	3.5
GPT-2	0.6
SimHash	2.0.0
LCS	–

4.2 *Experimental Design*

We performed four experiments in this section. Experiment 1 has 4 example sentences, and each sentence will select the keyword and then use the keyword to retrieve the sample program from the semantic database. The second experiment was to generate 100 programs separately from each sample program. Then check the code similarity between the sample program and the generated program, and verify whether the execution results of the generated program and the sample program are consistent. And analyze the performance of the generated program. The third experiment is to analyze the execution speed of the whole system. Experiment 4 explains the model.

This study established a semantic database for the experiment. XAMPP [22] created the keywords, example program names, example program paths, generated model paths, and other tables in the database, as shown in Fig. 20.



id	keyword	program-name	program-path	checkpoint-path	packet-program
0	weather	Web-Crawler	D:/	D:/checkpoint	D:/packet-program
1	neural	Neuralnetwork	D:/	D:/checkpoint	D:/packet-program
2	piano	Music	D:/	D:/checkpoint-path	D:/packet-program
3	video	Makevideo	D:/	D:/checkpoint-path	D:/packet-program

Figure 20: Table of four sample programs

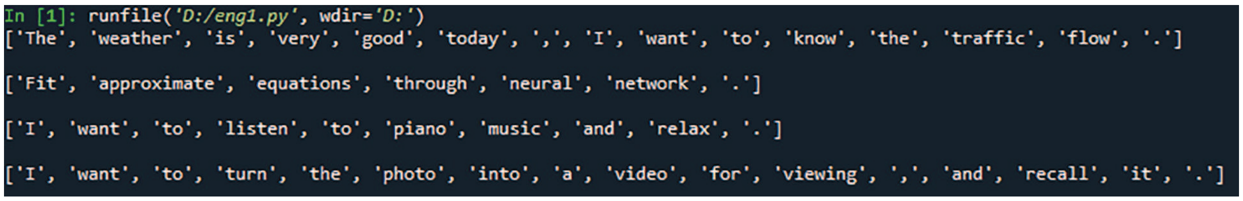
4.3 *Experimental Results*

4.3.1 *Experiment 1*

In Experiment 1, NLTK will be used to segment words from four input example sentences and select the appropriate keywords accordingly. Experiment 1 adopted four example sentences, as listed in Table 5. The results of word segmentation using NLTK have shown in Fig. 21.

Table 5: Example sentences

Case	Sentence content
Example 1	The weather is very good today, I want to know the traffic flow.
Example 2	Fit approximate equations through neural network.
Example 3	I want to listen to piano music and relax.
Example 4	I want to turn the photo into a video for viewing, and recall it.



```
In [1]: runfile('D:/eng1.py', wdir='D:')
['The', 'weather', 'is', 'very', 'good', 'today', ',', 'I', 'want', 'to', 'know', 'the', 'traffic', 'flow', '.']

['Fit', 'approximate', 'equations', 'through', 'neural', 'network', '.']

['I', 'want', 'to', 'listen', 'to', 'piano', 'music', 'and', 'relax', '.']

['I', 'want', 'to', 'turn', 'the', 'photo', 'into', 'a', 'video', 'for', 'viewing', ',', 'and', 'recall', 'it', '.']
```

Figure 21: Screenshot of NLTK word segmentation

The keywords have the corresponding sample programs precisely found and pick-up from the semantic database where the corresponding sample programs have entitled Web-Crawler, Neuralnetwork, Music, and Makevideo, as listed in Table 6. The sample programs in this study are all obtained from Github. The sample program in Example 1 is related to web crawlers [23], and the corresponding keywords are weather and traffic. Sample program 1 is to grab the corresponding data on the Internet, get the weather forecast from the weather center, and automatically assign the traffic jam spots on Google Maps. Next, in the example program of Example 2, the corresponding keyword is "equation, neural, network" [24] related to neural network applications. The primary purpose of the example program is to find an approximate equation by training a neural network. Third, in the sample program of Example 3, the corresponding keywords are piano and music, which is related to the program that generates music [25]. The webcam programming goal is to generate a short piece of piano music automatically. Finally, in Example 4, the keywords corresponding to the sample program are photo and video. This program can convert photos into videos for users to watch [26].

Table 6: The list of example programs in Experiment 1

Case	Extracted keywords from example sentence	Sample program
Example 1	weather, traffic	Web-Crawler
Example 2	equations, neural, network	Neuralnetwork
Example 3	piano, music	Music
Example 4	photo, video	Makevideo

4.3.2 Experiment 2

Experiment 2 with four sample programs implements in a single GPU workstation. This experiment first imported four sample programs into GPT-2, MASS, and BART to generate the preliminary programs. In Appendix, we have demonstrated a few samples of the generated preliminary programs.

After each sample program generates 100 preliminary programs, the next is to check the code similarity using the variational simhash (VSH) algorithm. We set a qualification level with the pass ratio of code similarity greater than or equal to 90%. Figs. 22–25 show the pass ratio of the generated preliminary programs. We have selected some of them with a higher pass ratio (≥90%) called qualified programs.

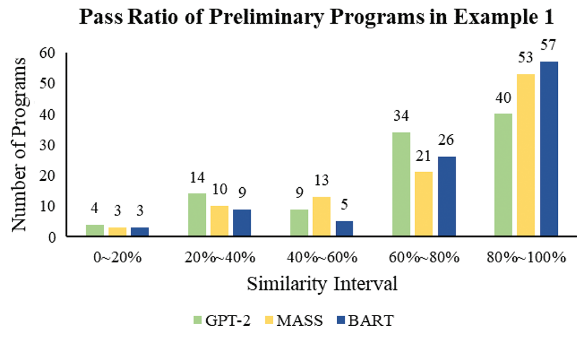


Figure 22: The pass ratio of the preliminary programs in Example 1

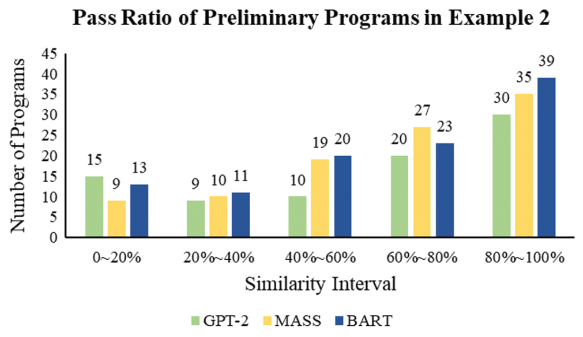


Figure 23: The pass ratio of the preliminary programs in Example 2

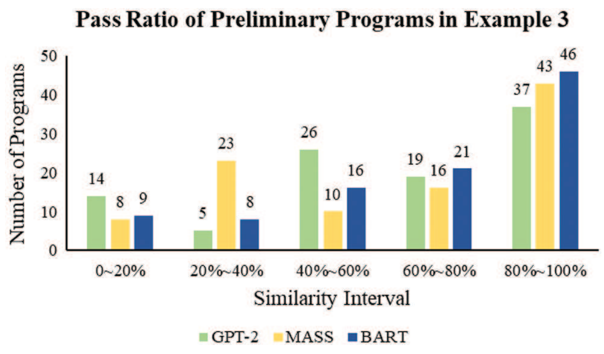


Figure 24: The pass ratio of the preliminary programs in Example 3

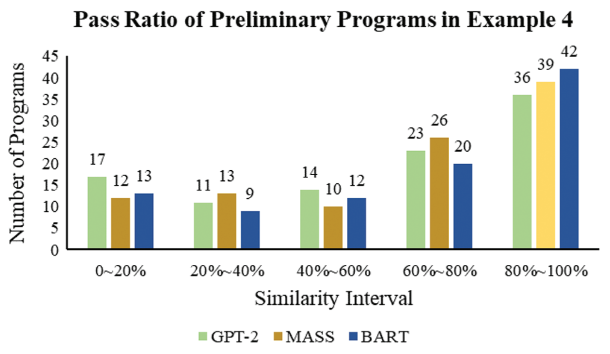
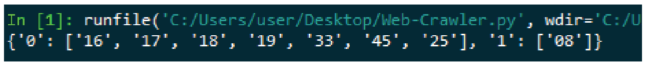


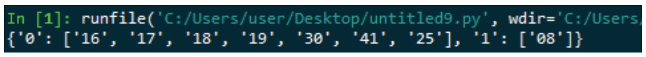
Figure 25: The pass ratio of the preliminary programs in Example 4

The third is to compile every qualified program. Once any program has complied successfully, we execute that program immediately. As shown in Figs. 26–29, PLCS will check the results of the successfully executed programs, as listed in Table 7. This PLCS is to find the conformity between the execution result of the sample program and the qualified program. The one with the highest compliance was called the pocket program.



```
In [1]: runfile('C:/Users/user/Desktop/Web-Crawler.py', wdir='C:/U
{'0': ['16', '17', '18', '19', '33', '45', '25'], '1': ['08']}
```

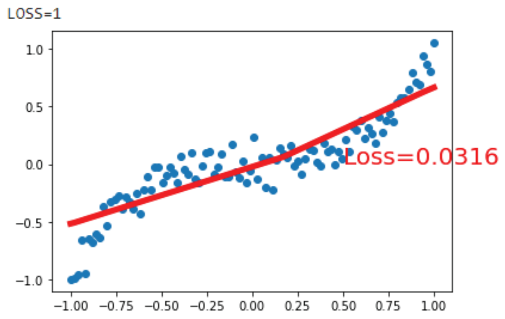
(a) Execution result of sample program 1



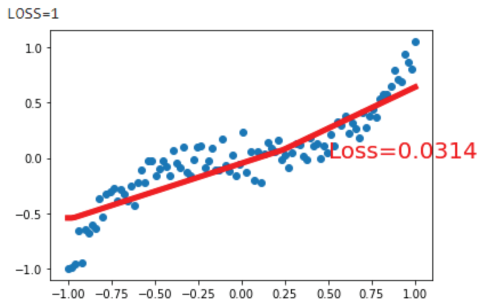
```
In [1]: runfile('C:/Users/user/Desktop/untitled9.py', wdir='C:/Users
{'0': ['16', '17', '18', '19', '30', '41', '25'], '1': ['08']}
```

(b) Execution result of newly generated program 1

Figure 26: Execution result of Example 1

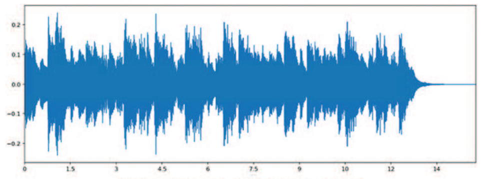


(a) Execution result of sample program 2

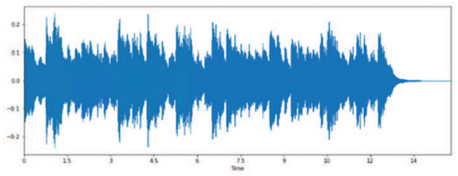


(b) Execution result of newly generated program 2

Figure 27: Execution result of Example 2

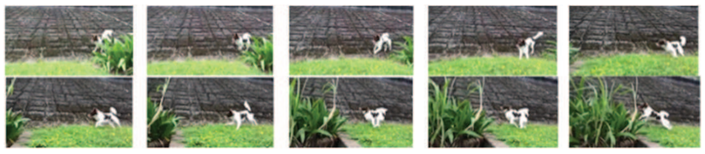


(a) Execution result of sample program 3

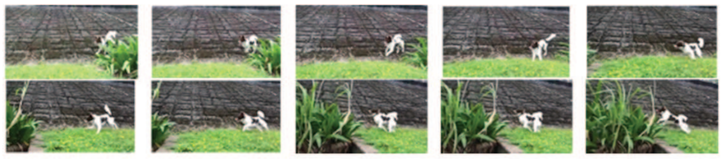


(b) Execution result of newly generated program 3

Figure 28: Execution result of Example 3



(a) Execution result of sample program 4



(b) Execution result of newly generated program 4

Figure 29: Execution result of Example 4

Table 7: PLCS conformity according to the number of identical codes (unit: %)

	Case											
Subject	Example 1			Example 2			Example 3			Example 4		
	GPT-2	MASS	BART	GPT-2	MASS	BART	GPT-2	MASS	BART	GPT-2	MASS	BART
Sample program	62	62	62	35218	35218	35218	162964	162964	162964	68087	68087	68087
Generated program	62	63	62	36513	36102	35783	188218	179341	179376	66537	66983	67210
Identical codes	61	61	61	35218	35195	34218	161998	160301	163911	65894	65912	66548
PLCS conformity	98.38	97.60	98.38	98.19	98.69	96.3	92.25	93.65	95.75	97.89	97.59	98.37

Finally, we have evaluated the performance of the proposed approaches, including VSH and PLCS algorithms, according to the execution result of sample programs and their respective pocket programs, as listed in Tables 8 and 9. Table 8 shows the number of code lines reduced between the sample program and the pocket program in four cases where the minimum number of code lines either in the sample program or the pocket program could be out of GPT-2, MASS, or BART. The proposed approach can reduce the number of code lines by 28.22% and program execution time by 30.98% on average. As a result, the proposed approach in this study can outperform the previous method published in 2021 [7].

Table 8: Number of code lines reduction

	Case			
Subject	Example 1	Example 2	Example 3	Example 4
Sample program	291	152	174	147
Pocket program	169* (174^)	123* (128^)	137# (146^)	102* (111^)
Reduction ratio (%)	41.92* (40.34^)	19.08* (15.78^)	21.26# (16.09^)	30.61* (24.48^)
Average reduction ratio (%)	28.22 (27.21^)			

Note: p.s. abbreviated symbol ^: GPT-2, #: MASS, and *: BART and parenthesis () indicating the minimum number of code lines of a sample program.

Table 9: Program execution time reduction (unit: second)

	Case			
Subject	Example 1	Example 2	Example 3	Example 4
Sample program	8.35	10.57	14.58	12.43
Pocket program	6.04* (6.97^)	6.89* (7.13^)	10.56# (11.71^)	7.97* (8.92^)
Reduction ratio (%)	27.66* (16.59^)	34.81* (32.54^)	27.57# (19.68^)	35.88* (28.23^)
Average reduction ratio (%)	30.98 (24.62^)			

Note: p.s. abbreviated symbol ^: GPT-2, #: MASS, and *: BART and parenthesis () indicating the minimum execution time of a sample program.

4.3.3 Experiment 3

In this study, the proposed method makes the generated programs produced more efficient and increases the system's speed. The first part is to improve code similarity comparison using the variational simhash (VSH) algorithm that can reduce the number of qualified programs. Reducing the number of qualified programs deducts the time required to compile all qualified programs. Compared with simhash (SH) algorithm, the proposed one obtained fewer qualified programs, as shown in Table 10. Table 10 shows that the reduction ratio of the number of qualified programs is up to 22.08%.

Table 10: Comparison of the number of qualified programs produced

Method	Example 1			Example 2			Example 3			Example 4		
	GPT-2	MASS	BART	GPT-2	MASS	BART	GPT-2	MASS	BART	GPT-2	MASS	BART
SH	23	34	41	25	29	32	28	37	42	31	36	40
VSH	19	27	33	19	21	23	22	29	32	25	29	31
Reduction ratio (%)	17.39	20.58	19.51	24.00	27.58	28.12	21.43	21.62	23.81	19.35	19.44	22.50
Average reduction ratio (%)	22.11											

Next, the proposed PLCS can perform the conformity check of the program execution results faster than the traditional LCS, as shown in Table 11. Therefore, it makes the system run rapidly. In Table 11, users pick up the best-performing pocket programs generated from GPT-2, MASS, or BART in Examples 1, 2, 3, and 4. Then users compare the conformity check using LCS and PLCS according to the number of string comparisons and its execution time. As a result, PLCS can reduce the number of character comparisons by 31.82% and the execution time shortened by 32.39%.

In contrast, the previous work [7] employed GPT-2 model, simhash algorithm, and LCS algorithm.

Table 11: Comparison of the conformity check

Case	The number of string comparisons in LCS	The number of string comparisons in PLCS	Reduction ratio (%)	Time for string comparisons in LCS (s)	Time for string comparisons in PLCS (s)	Reduction ratio (%)
Example 1	3721	3721	0.00	0.0030	0.0030	0.00
Example 2	1,240,307,524	937,040,324	24.45	392	279	28.83
Example 3	26,557,265,296	17,808,785,296	32.94	7839	5268	32.80
Example 4	4,635,839,569	3,368,007,569	27.35	894	622	30.43
Average	8,108,354,027.5	5,528,459,227.5	31.82	2,281.25	1542.25	32.39

The proposed approach in this study uses GPT-2, MASS, or BART models, variational simhash algorithm, and PLCS algorithm. Tables 10 and 11 confirm that the proposed method improves the efficiency of producing the generated programs and the execution speed of the conformity check. Then, users can calculate the entire process's execution time t_{total} in Eq. (17), where $t_{segmentation}$ represents the time of selecting keywords after word segmentation using NLTK, $t_{sample\ program\ retrieval}$ stands for the time of searching the corresponding sample program in the semantic database, $t_{code\ transformation}$ is the time of producing the newly generated program from the code transformation models, $t_{code\ similarity\ check}$ denotes the time for checking code similarity, $t_{compling\ qualified\ program}$ expresses the time for compiling all qualified programs, $t_{conformity\ check}$ indicates the time of the consistency check of execution result, and $t_{pocket\ program\ execution}$ refers to the time to run a pocket program. Finally, users must use the code transformation model to estimate the time taken for the entire process of code transformation, as listed in Table 12. Eq. (18) has defined a performance index to express the speedup factor between two different methods. As a result, the proposed approach outperforms the method mentioned in the previous work, increasing the speed of the entire code transformation process up to 1.38 times.

$$t_{total} = t_{segmentation} + t_{sample\ program\ retrieval} + t_{code\ transformation} + t_{code\ similarity\ check} + t_{compling\ qualified\ program} + t_{conformity\ check} + t_{pocket\ program\ execution} \tag{17}$$

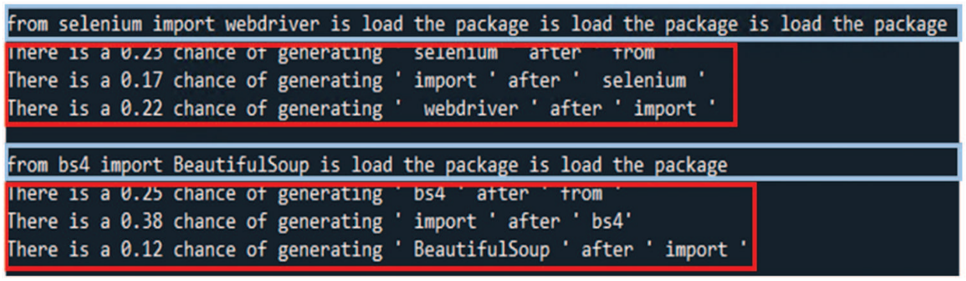
$$PI_{proposed} = \frac{\frac{1}{t_{total\_proposed}}}{\frac{1}{t_{total\_previous}}} \tag{18}$$

Table 12: Execution time of the entire process and performance index

Case	$t_{total\_previous}$ (s)	$t_{total\_proposed}$ (s)	PI
Example 1	2635.34	2633.62	1.00
Example 2	3045.50	2922.02	1.04
Example 3	10,478.24	5655.76	1.85
Example 4	3570.98	3098.50	1.15
Average	4932.52	3577.48	1.38

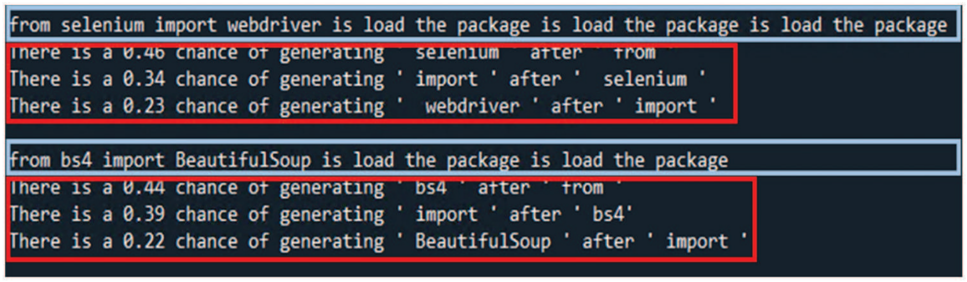
4.3.4 Experiment 4

This study uses LIME to explain the decision-making from AI models or algorithms such as GPT-2, MASS, BART, simhash, variational simhash, LCS, and PLCS. First, users applied LIME to interpret the outcomes of the decisions made from three pre-trained code transformation models, GPT-2, MASS, and BART. Given sample program 1, GPT-2, MASS, or BART produced the newly generated programs and then sent them into LIME to obtain the explainable results, as shown in Figs. 30–32. The results show the effect of each line of the program and its probability of being generated. It shows that pre-trained code transformation models can decide what code should not be generated, thus making the code transformation process more efficient. As a result, there is no difference in the code transformation results among the three models mentioned above.



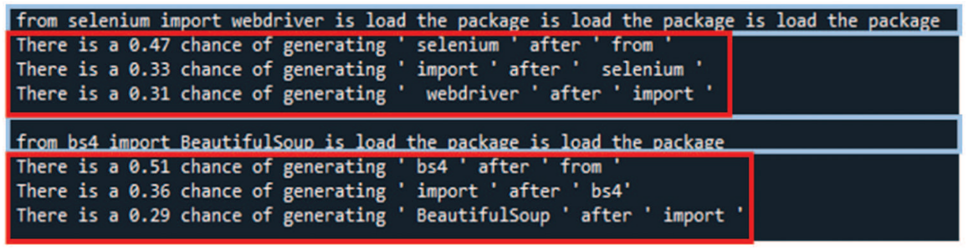
```
from selenium import webdriver is load the package is load the package is load the package
There is a 0.23 chance of generating ' selenium ' after ' from '
There is a 0.17 chance of generating ' import ' after ' selenium '
There is a 0.22 chance of generating ' webdriver ' after ' import '

from bs4 import BeautifulSoup is load the package is load the package
There is a 0.25 chance of generating ' bs4 ' after ' from '
There is a 0.38 chance of generating ' import ' after ' bs4'
There is a 0.12 chance of generating ' BeautifulSoup ' after ' import '
```

Figure 30: LIME explains the results produced by GPT-2



```
from selenium import webdriver is load the package is load the package is load the package
There is a 0.46 chance of generating ' selenium ' after ' from '
There is a 0.34 chance of generating ' import ' after ' selenium '
There is a 0.23 chance of generating '  webdriver ' after ' import '

from bs4 import BeautifulSoup is load the package is load the package
There is a 0.44 chance of generating ' bs4 ' after ' from '
There is a 0.39 chance of generating ' import ' after ' bs4'
There is a 0.22 chance of generating ' BeautifulSoup ' after ' import '
```

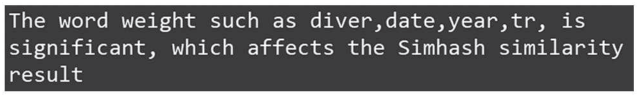
Figure 31: LIME explains the results produced by MASS



```
from selenium import webdriver is load the package is load the package is load the package
There is a 0.47 chance of generating ' selenium ' after ' from '
There is a 0.33 chance of generating ' import ' after '  selenium '
There is a 0.31 chance of generating '  webdriver ' after ' import '

from bs4 import BeautifulSoup is load the package is load the package
There is a 0.51 chance of generating ' bs4 ' after ' from '
There is a 0.36 chance of generating ' import ' after ' bs4'
There is a 0.29 chance of generating ' BeautifulSoup ' after ' import '
```

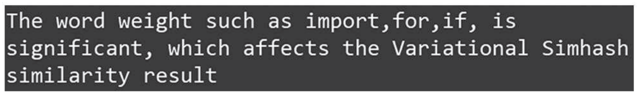
Figure 32: LIME explains the results produced by BART

Next, users applied LIME to explain the decision-making from the algorithms of code similarity check, both simhash and variational simhash algorithms. Given the preliminary programs and the corresponding sample program, simhash or variational simhash produced the newly qualified programs and then sent them into LIME to obtain the explainable results, as shown in Figs. 33–34. The weights of the words in a code line affect the result of the code similarity check. Finally, given the qualified programs and the corresponding sample program, LCS and PLCS produced the comparison of the execution results of the sample program and the pocket program. They then sent them into LIME to obtain the explainable results, as shown in Figs. 35 and 36. Consequently, the ASCII or binary code length affects the execution time significantly.



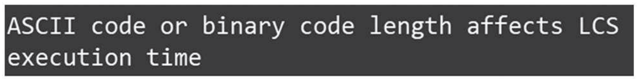
```
The word weight such as diver,date,year,tr, is
significant, which affects the Simhash similarity
result
```

Figure 33: LIME explains the results produced by simhash



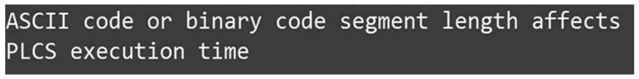
```
The word weight such as import,for,if, is
significant, which affects the Variational Simhash
similarity result
```

Figure 34: LIME explains the results produced by variational simhash



```
ASCII code or binary code length affects LCS
execution time
```

Figure 35: LIME explains the results produced by LCS



```
ASCII code or binary code segment length affects
PLCS execution time
```

Figure 36: LIME explains the results produced by LCS

This study uses LIME to explain the AI model or algorithm decision-making and let people learn how the system works to optimize the algorithm and increase the overall efficiency. This study complies consistently with the European Parliament-issued Ethics Guidelines for Trustworthy AI, proving that this study is trustworthy.

4.4 Discussion

In terms of code transformation models GPT-2, MASS, and BART introduced in Section 2, this study has validated that we used a pre-trained model for the transfer learning technique to train a generative program model running two passes through code transformation models. Then it can produce higher-performance generated programs newly. However, the followed verification process in the test phase will encounter the time-consuming problem for the checks of code similarity and execution result conformity, mainly a considerable amount of output data taking a long time. Therefore, the proposed variational simhash algorithm can appropriately adjust the weighted values of keywords or reserved words of Python programming. The accuracy of the code similarity check can increase by 24%, superior to the simhash algorithm, as described in Section 3.2. Moreover, compared with LCS described in Section 3.3, the proposed PLCS algorithm can reduce the number of comparisons by 31.82% of conformity checks and speed up the computation by 32.39%. Finally, in the experimental results in Section 4, the performance evaluation outcomes have testified that both proposed approaches can effectively speed up the entire code transformation process significantly.

In Section 4.3.2, according to the execution results of use cases of text, pictures, and videos, Tables 7 and 8 have shown that the performance of BART is better than that of GPT-2 and MASS. Regarding the execution result of the use case of speech, the performance of MASS is better than that of GPT-2 and BART. In Section 4.3.3, compared with simhash algorithm, the proposed variational simhash algorithm can reduce the number of qualified programs effectively, as listed in Table 10. Accordingly, it has taken less time to compile all qualified programs. Compared with LCS algorithm, the proposed PLCS algorithm can reduce the number of string comparisons for performing the conformity check of program execution results, as shown in Table 11. Thus, it has considerably reduced the time for the outcome verification and speeds up the entire code transformation process. In Section 4.3.4, this study uses LIME to interpret the decision-making mechanism of applied algorithms or models. Figs. 30–36 enable readers to understand better how to produce the outputs from algorithms or models, such as GPT-2, MASS, BART, SH, VSH, LCS, and PLCS.

Regarding limitations, because this study only designs variational simhash algorithm for Python programming code, this algorithm for other programming languages is not applicable. Users must recode such an algorithm according to the keywords or reserved words of the designated programming language separately. Furthermore, the proposed approach PLCS in this study only has divided a long string into five different lengths of a segment to implement a segment-by-segment comparison between two strings. If there are more than five different lengths of a segment to carry out the PLCS algorithm, it may be possible to get a faster job execution. So far, there are not many keywords and sample programs stored in the semantic database. As more and more keywords and sample programs have stored in the semantic database, it will result in slower keyword searching in the semantic database and poor overall performance of the code transformation process.

5 Conclusion

The main contribution of this paper is to improve the execution efficiency on both the similarity check of the program's code and the conformity check of program execution results, which can speed up the entire code transformation process significantly. In other words, the proposed approaches have achieved the objective of this study to implement a high-efficient code transformation process by reducing the execution time considerably. Compared with the previous work, the proposed approaches can significantly speed up the entire code transformation process by 1.38 times. Meanwhile, explainable AI techniques can also interpret the decision-making of AI models. Here we use LIME to let people learn how the system works to optimize the algorithm and increase the overall efficiency.

In future work, we must devote ourselves to database searching and variational simhash algorithm improvements. Regarding database searching, users can develop deep learning methods to optimize data storage to rapidly retrieve keywords and sample or pocket programs in a semantic database. For the variational simhash algorithm, we should consider extending the variational simhash algorithm to several popular programming languages except for Python. Thus, the above improvements can hopefully conquer the limitations stated in the discussion section.

Data Availability: Readers can find the Sample Program.zip data used to support this study's findings deposited in the https://drive.google.com/file/d/1SPfMmKr43bQDvxU5iY9O1jpab7H-GUGn/view? The article has included the sample sentence data used to support the findings of this study.

Author Contributions: B.R.C. and H.-L.C. conceived and designed the experiments; H.-F.T. collected the experimental dataset and proofread the paper; B.R.C. wrote the paper.

Funding Statement: This work is fully supported by the Ministry of Science and Technology, Taiwan, under Grant Nos. MOST 111-2221-E-390-012 and MOST 111-2622-E-390-001.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

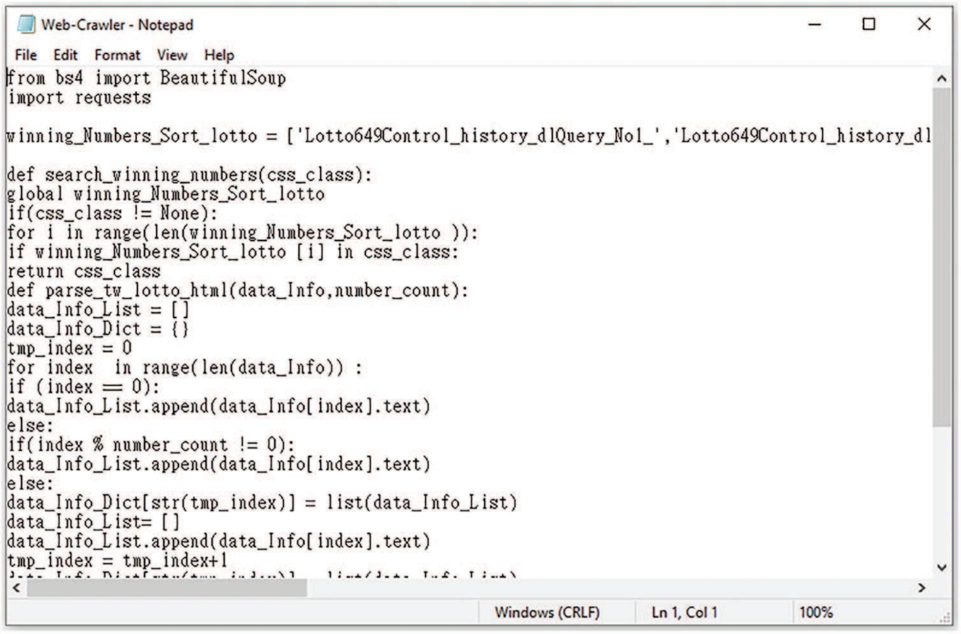
References

1. Chakaravarthy, R. V., Jiang, H. (2020). Special session: XTA: Open source extensible, scalable and adaptable tensor architecture for AI acceleration. *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pp. 53–56. Hartford, CT, USA.
2. Kumar, S. D., Subha, D. P. (2019). Prediction of depression from EEG signal using long short term memory (LSTM). *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, pp. 1248–1253. Tirunelveli, India.
3. Önder, M., Akgül, Y. S. (2020). Automatic generation of matching clothes design using generative adversarial networks. *2020 28th Signal Processing and Communications Applications Conference (SIU)*, pp. 1–4. Gaziantep, Turkey.
4. Bird, S., Klein, E., Loper, E. (2010). Natural language processing with python, analyzing text with the natural language toolkit. *Language Resources and Evaluation, 44(4),* 421–424.

5. Radford, A., Narasimhan, K., Salimans, T., Sutskever, I. (2019). Improving language understanding by generative pre-training. https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf.
6. Yanagi, Y., Orihara, R., Sei, Y., Tahara, Y., Ohsuga, A. (2020). Fake news detection with generated comments for news articles. *2020 IEEE 24th International Conference on Intelligent Engineering Systems (INES)*, pp. 85–90. Reykjavík, Iceland.
7. Chang, B. R., Tsai, H. F., Su, P. W. (2021). Code transform model producing high-performance program. *Computer Modeling in Engineering & Science, 129(1),* 253–277. DOI 10.32604/cmes.2021.015673.
8. Song, K., Tan, X., Qin, T., Lu, J., Liu, T. Y. (2019). Mass: Masked sequence to sequence pre-training for language generation. *arXiv preprint arXiv: 1905.02450.*
9. Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A. et al. (2019). Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv: 1910.13461.*
10. Yuan, Y., Li, R., Wang, Y., Cao, T., Yang, J. et al. (2020). Application of the maintenance text data of transformers based on SimHash and Hamming distance algorithm. *2020 IEEE International Conference on High Voltage Engineering and Application (ICHVE)*, pp. 1–4. Beijing, China.
11. Qin, M. (2018). Hamming-distance-based binary representation of numbers. *2018 Information Theory and Applications Workshop (ITA)*, pp. 1–9. San Diego, CA, USA: IEEE.
12. Chen, X., Peng, A., Tang, B. (2020). Automatic radio map adaptation for wifi fingerprint positioning systems. *2020 5th International Conference on Communication, Image and Signal Processing (CCISP)*, pp. 64–69. Chengdu, China.
13. Burghardt, J. (2021). Longest common subsequence problem. https://en.wikipedia.org/wiki/Longest_common_subsequence_problem.
14. Ribeiro, M. T., Singh, S., Guestrin, C. (2016). "Why should i trust you?": Explaining the predictions of any classifier. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 1135–1144. San Francisco, CA, USA.
15. Chou, H. L. (2021). Test1. https://github.com/m1085504/Data-exsaple/blob/main/test.
16. Chou, H. L. (2021). Test2. https://github.com/m1085504/Data-exsaple/blob/main/test1.
17. Chou, H. L. (2021). Exchange-rate. https://github.com/m1085504/Data-exsaple/blob/main/Exchange-Rate.
18. Chou, H. L. (2021). Picture. https://github.com/m1085504/Data-exsaple/blob/main/picture.
19. Chou, H. L. (2021). Voice. https://github.com/m1085504/Data-exsaple/blob/main/voice.
20. Chou, H. L. (2021). Video. https://github.com/m1085504/Data-exsaple/blob/main/video.
21. Mantor, M. (2019). 7NM "NAVI" GPU–A GPU built for performance and efficiency. *2019 IEEE Hot Chips 31 Symposium (HCS)*, Cupertino, CA, USA.
22. Agustin, F., Kurniawan, H., Yusfrizal, Y., Ummi, K. (2019). Comparative analysis of application quality between AppServ and XAMPP webserver using AHP based on ISO/IEC 25010:2011. *2018 6th International Conference on Cyber and IT Service Management (CITSM)*. Parapat Nort Sumatera, Indonesia.
23. Lin, J. W. (2020). Web-crawler. https://github.com/jwlin/web-crawler-tutorial.
24. Chou, H. L. (2021). Network. https://github.com/m1085504/Data-exsaple/blob/main/Network.
25. Chou, H. L. (2021). Music. https://github.com/m1085504/Data-exsaple/blob/main/Mucis.
26. Chou, H. L. (2021). Makevideo. https://github.com/m1085504/Data-exsaple/blob/main/Makevideo.

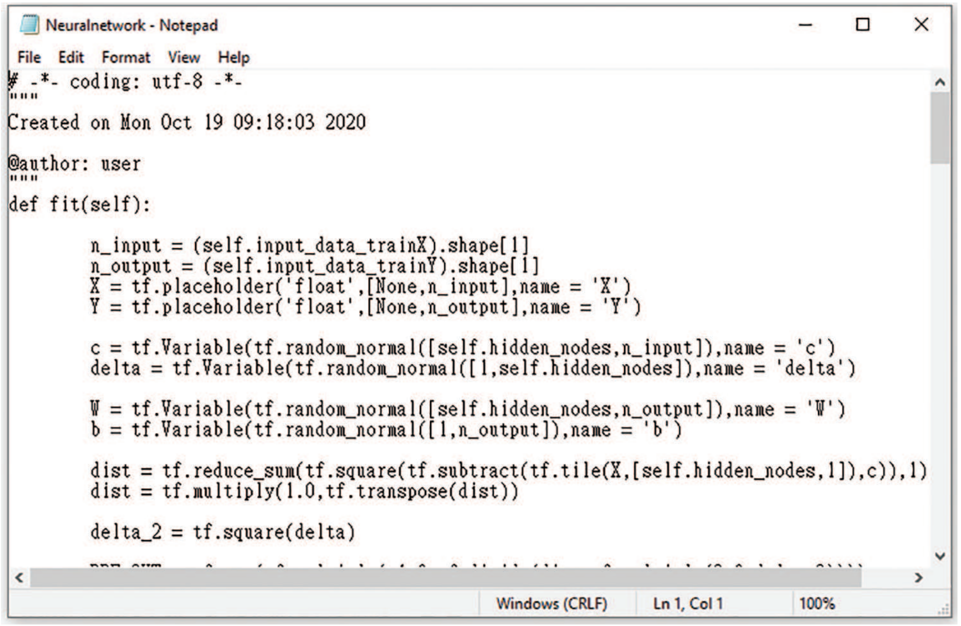
Appendix

In the Experiment 2, samples of the generated programs are shown in Figs. 37–40.



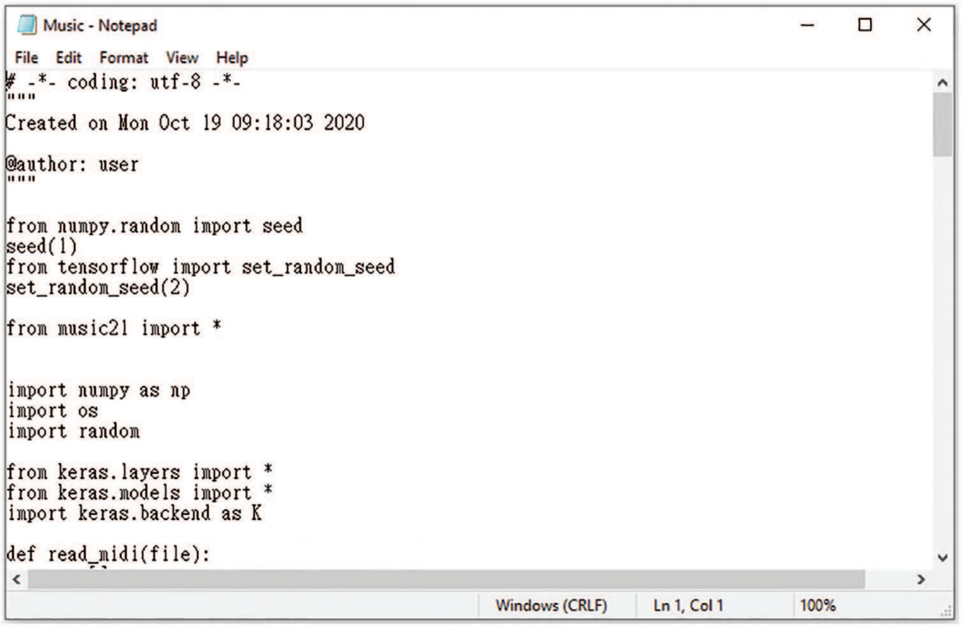
```
from bs4 import BeautifulSoup
import requests

winning_Numbers_Sort_lotto = ['Lotto649Control_history_dlQuery_No1_','Lotto649Control_history_dl

def search_winning_numbers(css_class):
global winning_Numbers_Sort_lotto
if(css_class != None):
for i in range(len(winning_Numbers_Sort_lotto )):
if winning_Numbers_Sort_lotto [i] in css_class:
return css_class
def parse_tw_lotto_html(data_Info,number_count):
data_Info_List = []
data_Info_Dict = {}
tmp_index = 0
for index  in range(len(data_Info)) :
if (index == 0):
data_Info_List.append(data_Info[index].text)
else:
if(index % number_count != 0):
data_Info_List.append(data_Info[index].text)
else:
data_Info_Dict[str(tmp_index)] = list(data_Info_List)
data_Info_List= []
data_Info_List.append(data_Info[index].text)
tmp_index = tmp_index+1
```

Figure 37: Sampled preliminary program associated with sample program 1 in Experiment 2



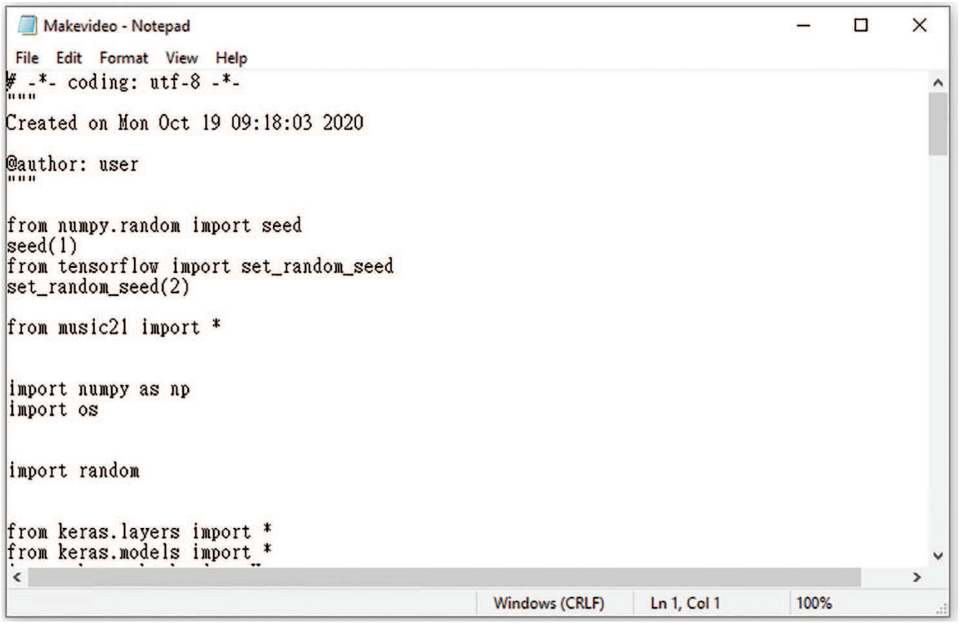
```
# -*- coding: utf-8 -*-
"""
Created on Mon Oct 19 09:18:03 2020

@author: user
"""
def fit(self):

        n_input = (self.input_data_trainX).shape[1]
        n_output = (self.input_data_trainY).shape[1]
        X = tf.placeholder('float',[None,n_input],name = 'X')
        Y = tf.placeholder('float',[None,n_output],name = 'Y')

        c = tf.Variable(tf.random_normal([self.hidden_nodes,n_input]),name = 'c')
        delta = tf.Variable(tf.random_normal([1,self.hidden_nodes]),name = 'delta')

        W = tf.Variable(tf.random_normal([self.hidden_nodes,n_output]),name = 'W')
        b = tf.Variable(tf.random_normal([1,n_output]),name = 'b')

        dist = tf.reduce_sum(tf.square(tf.subtract(tf.tile(X,[self.hidden_nodes,1]),c)),1)
        dist = tf.multiply(1.0,tf.transpose(dist))

        delta_2 = tf.square(delta)
```

Figure 38: Sampled preliminary program associated with sample program 2 in Experiment 2



```
# -*- coding: utf-8 -*-
"""
Created on Mon Oct 19 09:18:03 2020

@author: user
"""

from numpy.random import seed
seed(1)
from tensorflow import set_random_seed
set_random_seed(2)

from music21 import *


import numpy as np
import os
import random

from keras.layers import *
from keras.models import *
import keras.backend as K

def read_midi(file):
```

Figure 39: Sampled preliminary program associated with sample program 3 in Experiment 2



```
# -*- coding: utf-8 -*-
"""
Created on Mon Oct 19 09:18:03 2020

@author: user
"""

from numpy.random import seed
seed(1)
from tensorflow import set_random_seed
set_random_seed(2)

from music21 import *


import numpy as np
import os


import random


from keras.layers import *
from keras.models import *
```

Figure 40: Sampled preliminary program associated with sample program 4 in Experiment 2