Penerbit Akademia Baru

Journal of Advanced Research Design

Journal homepage:
https://akademiabaru.com/submit/index.php/ard
ISSN: 2289-7984

A Comparative Evaluation of Transformers in Seq2Seq Code Mutation: Non-Pre-trained Vs. Pre-trained Variants

Loh Zheung Yik[1], Wan Mohd Nasir Wan Kadir[1,*], Noraini Ibrahim[1]

[1] Department of Software Engineering, Faculty of Computing, Universiti Teknologi Malaysia, 81310 Skudai, Johor, Malaysia

ARTICLE INFO

Article history:
Received 22 March 2024
Received in revised form 21 September 2024
Accepted 16 December 2024
Available online 31 December 2024

Keywords:
Mutation testing; transformer; seq2seq; code mutation; mutant; trivial mutant

ABSTRACT

Mutation testing (MT) is a gold standard way to assess the efficacy of software test suites. However, the accuracy of mutation score is affected by the presence of trivial mutants which can be "killed" by even the simplest and most basic test suites. Since the existence of trivial mutants is due to the fixed set of mutation operators that constraints the complexity of code mutations, state-of-the-art recurrent neural network (RNN) model is used for sequence-to-sequence (seq2seq) code mutation without relying on mutation operators. However, the quality of the produced mutants is affected by the limitation of RNN in interpreting the relationships between far-apart tokens of the code to be mutated. Transformers that do not have this limitation, have superseded RNN in seq2seq machine translation domains such as natural language processing (NLP). However, to the best of our knowledge, there is still no research that investigates the performance of transformers in seq2seq code mutation. This paper presents a comparison study that involves different variants of the non-pre-trained transformers, the transformers pre-trained with source code, the transformers pre-trained with natural language, and the state-of-the-art RNN model in seq2seq code mutation. The results show that transformers pre-trained with source code, especially CodeT5, demonstrated the best performance, achieving an average character n-gram F-score (CHRF) of 82.89 and superior code mutation complexity. Since the performance of transformers in seq2seq code mutation has not been previously investigated, the primary contribution of this paper is the best performing transformer for seq2seq code mutation. It establishes the foundation for the future research that proposes an integrated solution which addresses both the high-cost problem and the inaccurate mutation score problem of MT simultaneously, unlike existing solutions which only tackle one of the MT problems and give rise to other MT problems.

1. Introduction

Software test suite quality in terms of sufficiency and fault-detection capability, needs to be emphasized because test suites play a crucial role in guiding the software testing process. To measure software test suite quality, mutation score is a better metric than code coverage because it verifies whether the program states of the software under test (SUT) are indeed reachable by propagating the injected faults to the observable output, while code coverage only verifies if a part of the SUT's

* *Corresponding author.*
E-mail address: wnasir@utm.my

https://doi.org/10.37934/ard.123.1.4565

code is executed by the test suite [1]. Mutation testing (MT) needs to be conducted to calculate mutation score. However, MT is affected by problems such as the presence of trivial mutants which affect the accuracy of mutation score [2, 3]. Trivial mutants are mutants that have injected faults which can be easily detected or "killed" by any test suites including the simple and lacking ones. This problem has caused the low adoption rate of MT in the industry and poses a threat to the validity of test suite quality improvement approaches proposed by the academia which use mutation score as a validation metric [4]. Ideally, the injected faults should resemble real faults made by software developers for effective assessment of software test suite efficacy during MT [5, 6]. It is difficult to produce mutants with realistic faults if the degree of code mutation is constrained by a fixed set of mutation operators [7]. Citing from natural language processing (NLP) and machine translation approaches by Tufano *et at.,* [8] uses recurrent neural network (RNN) to mutate code in a sequence-to-sequence (seq2seq) manner without relying on mutation operators. However, the quality of the mutants is affected by the limitation of RNN in capturing relationships between tokens that are far apart in the input code sequence [9].

After the introduction of transformers, there are researchers in the NLP and machine translation field that adopt transformers instead of RNNs, as reported by Stefenon *et al.,* [10] and Ghani *et al.,* [11]. This is because transformers can capture complex relationships between tokens in the input sequence, even those that are far apart, by interpreting the sequence tokens simultaneously [10, 12]. In the case of code mutation for MT, there is an existing approach that uses transformer to produce mutants [13]. However, that transformer-based approach does not produce mutants with multiple code modifications with a single prediction in a seq2seq manner as the transformer only have encoder which only predicts a replacement token for the code sequence that have one removed token.

To date, there are many researchers that use pre-trained transformers such as CodeT5 and PLBART for seq2seq source code-related tasks such as code summarization and programming language translation [14-16]. However, to the best of our knowledge, there is no research that investigates the performance of non-pre-trained transformers and pre-trained transformers in generating mutants with realistic faults in a seq2seq manner. Meanwhile, existing research shows that are transformer that is pre-trained using natural language corpora performs better than transformer that is pre-trained using source code corpora in some source code-related tasks [17]. Hence, in this paper, we will investigate and compare the performances of original non-pre-trained transformer, transformers pre-trained with source code corpora, and transformers pre-trained with natural language corpora, in translating input code sequences into mutated code sequence for MT. The research questions of this paper are as follows;

RQ 1. Do the pre-trained transformers perform better than the non-pre-trained transformer and the state-of-the-art RNN in mutating code?
RQ 2. Does the type of pre-training data and the pre-training method influence the transformers in the downstream task of seq2seq code mutation?
RQ 3. What are the characteristics of the mutants produced by the non-pre-trained transformers, the transformers pre-trained with source code corpora, the transformers pre-trained with natural language, and the state-of-the-art RNN model?

Since the performance of transformers in seq2seq code mutation has not been previously investigated, the primary contribution of this paper is the best performing transformer for seq2seq code mutation. It establishes the foundation for the future research that proposes an integrated solution which addresses both the high-cost problem and the inaccurate mutation score problem of

MT simultaneously, unlike existing solutions which only tackle one of the MT problems and give rise to other MT problems. In other words, the research contribution can help to address the research gap identified in our previously published systematic literature review paper [18].

2. Background and Related Work

2.1 Mutation Testing (MT)

MT is a gold standard technique for verifying the efficacy of a software test suite in terms of sufficiency and fault-finding capability [19]. During conventional MT, the code of the SUT is modified to produce mutants, which are faulty versions of the SUT [19]. The nature of the code modification depends on the type of mutation operator that is applied. For example, applying the relational operator replacement (ROR) operator changes "<" token in the code to "<=", as shown in Figure 1. Then, the mutants are executed against the test suite and compared with the test execution results of the original SUT. If the test suite can differentiate between the original SUT and a mutant by producing different test outputs, then the mutant is considered killed [20]. Ideally, the injected faults should resemble real faults made by software developers for effective assessment of software test suite efficacy [5, 6].

After all mutants have been executed and compared with the original SUT, the mutation score, which measures the test suite's efficacy, is calculated [21]. The mutation score is the proportion of killed mutants among the non-equivalent mutants as shown in Eq. (1). Equivalent mutants are mutants which are impossible to be killed because they tend to produce the same output as the original unmutated SUT [22]. They need to be identified manually among the alive mutants and be discarded so that the mutation score will not become inaccurate. As for the mutants that remains alive after mutant execution, they can be used as a guideline to improve the test suite coverage.

$$Mutation\ score = \frac{Number\ of\ killed\ mutants}{Total\ number\ of\ mutants - Number\ of\ Equivalent\ Mutants} \tag{1}$$

2.2 Mutation Testing (MT) Problems

One of the problems that causes low adoption of MT in the software industry is the presence of unproductive trivial mutants. The case study by Petrovic *et al.,* [4] found out that software developers are reluctant to adopt MT because there are too many unproductive trivial mutants that cannot lead to test suite improvements. Trivial mutants are mutants that can be killed by any test suites including the ones that are simple and lacking [2, 3]. Killing a large number of trivial mutants can lead to high but misleading mutation score. The high mutation score does not really reflect that the test suite has high fault detection capability. Figure 1 shows an example of trivial mutants, if "<" in line 6 is mutated to become "<=", any test suites that involve with the execution of this code scope will encounter an out-of-bounds exception and causes an increase in mutation score. If many of such mutants exist in the mutant population, the mutation score accuracy will be undermined.

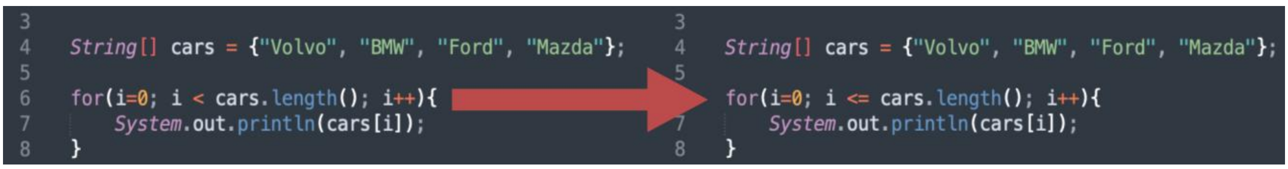

Fig. 1. Example of trivial mutants

The factors that cause the trivial mutant problem (TMP) includes the usage of first order mutants (FOM) which only have one code modification in each mutant. FOMs may not be able to simulate real faults that are usually complex. The empirical analysis by Gopinath *et al.,* [6] shows that typical software faults involve more than 3 tokens. Besides, the fixed set of mutation operators that don't have enough expressiveness to create realistic artificial faults will also cause TMP [7]. Another factor that causes TMP is the code mutation process that blindly choose the code parts to be mutated without considering the full context of the code that will be mutated [23].

Even for small sized SUT, it is possible to generate a large number of mutants and many of them are trivial mutants [24]. The large number of mutants which caused the MT process to become time consuming have led companies such as the large international safety critical system company interviewed by Vercacmmen *et al.,* [25] to refuse the adoption of MT. Even though the empirical analysis results show that MT will not be costly if only the productive non-trivial mutants are involved [4], however, it is difficult to find the useful subset of mutants among the large mutant population.

Existing solutions of TMP include the usage of higher order mutants (HOM) which have more than one code modification in each mutant [26]. HOM is said to be better at simulating the real software faults that are usually complex [26]. However, search strategies [27, 28] to find the useful subset of HOMs is required because the number of FOM combinations that can form HOMs is exponential while some HOMs can be as trivial as FOMs [29]. Useful HOMs are those that have artificial faults which cannot be simulated by any individual FOMs [29]. Moreover, the degree of code mutation found in HOMs is still limited by the fixed set of mutation operators [7]. Meanwhile, the approach proposed in this paper uses transformer to produce mutants in a seq2seq manner and does not require the usage of mutation operators.

To reduce the reliance on mutation operators, some researchers propose to mutate code by applying bug patterns extracted from bug reports [23]. In contrast to our approach proposed in this paper, this method does not involve deep learning to learn bug patterns or to decide the mutation location in the code. There are also ML-based approaches that can mutate code without involving mutation operators. For instance, Tufano *et al.,* [8] uses RNN which is trained using pairs of buggy code and related fixed code to mutate code in a seq2seq manner. However, RNNs are not very proficient at capturing relationships between tokens that are far apart in the code sequence [9], and as a result, they may produce mutants with syntax errors. Different from that RNN-based approach, our approach proposed in this paper uses transformers which can capture complex relationships between tokens in the input sequence, even those that are far apart, by interpreting the sequence tokens simultaneously. Degiovanni and Papadakis [13] adopt an encoder-only transformer to avoid the drawbacks of RNN. However, the encoder-only transformer only predicts a replacement token for the code sequence that have one removed token. In other words, the encoder-only transformer does not produce mutants with multiple mutated code parts in a seq2seq manner like the RNN. In contrary, the transformer-based approach proposed in this paper possess both encoder and decoder to mutate code in a seq2seq manner. The input is the code to be mutated while the output is the mutated code. Table 1 shows the comparison between the transformer-based approach proposed in this paper and the existing solutions of TMP.

2.3 Sequence to Sequence (seq2seq)

Sequence-to-sequence (seq2seq) is a machine learning (ML) field that involves the generation of an output sequence from an input sequence. It is widely implemented in NLP and time series data forecasting [11, 10]. Many researchers adopt RNN that can handle sequential data of varying lengths are used. However, since RNN is poor at handling the dependency between tokens that are located far apart from each other in the sequence, researchers opt to use RNN variants such as long short-term memory (LSTM) or gated recurrent unit (GRU) that have memory mechanisms so that it can perform better in interpreting the dependencies between far-apart tokens [30].

LSTM and GRU may not be able to achieve optimal accuracy if the prediction requires parallel interpretations of the tokens in the sequence because LSTM and GRU process tokens in a sequential manner. This has been caused by Chen *et al.,* [30] to adopt a dense network of simple recurrent units to address the parallelism problem. Meanwhile, there are also researchers who form RNN ensembles with other ML models such as graph convolutional network to increase accuracy [31].

Since the introduction of transformers by Vaswani *et al.,* [12], researchers began to adopt transformers for seq2seq learning tasks. Transformers have self-attention mechanism that allows them to interpret the tokens in the input sequence simultaneously. As a result, transformers can perform better than RNN in interpreting far-apart tokens in the input sequence while producing output.

2.4 Transformer

Similar to RNN, transformers also have encoder and decoder that allow it to perform seq2seq task. One notable application of Seq2seq transformer is ChatGPT which is a popular artificial intelligence-powered chatbot. Meanwhile, encoder-only transformers are used for non-seq2seq tasks, such as making decisions from time series input data [32]. The popularity of transformers has given rise to the existence of many pre-trained transformer models. Examples of pre-trained transformer models with both encoder and decoder, suitable for seq2seq learning, include CodeT5, PLBART, Pegasus, and Prophetnet [14, 15, 33, 34]. The pre-trained transformers have readily initialized weights that result from the pre-training process. The pre-trained transformer needs to be fine-tuned using domain specific dataset before they can be utilized for downstream tasks.

The pre-trained transformers differ with each other in terms of the type of data used for pre-training, pre-training methods and the way they process tokens in the input sequence. For instance, CodeT5 and PLBART are pre-trained with source code corpora while Pegasus and Prophetnet are pre-trained with natural language corpora. Unlike PLBART which treats the source code corpora similarly to how NLP pre-trained transformers treat the natural language corpora, CodeT5 labels the code tokens in the dataset as identifiers and non-identifiers during pre-training. Meanwhile, Prophetnet predicts n future tokens for the output and uses the information from these future tokens to predict additional future tokens for the output sequence.

Table 1
Comparison between the proposed transformer-based approach and the existing TMP solutions

Difference	Proposed Transformer-based Approach	RNN-based Approach by Tufano *et al.,* [8]	Masked token prediction using encoder-only transformer by Degiovanni and Papadakis [13]	Higher Order Mutation [26-29]
Way to generate mutants.	Generate mutants in seq2seq manner after learning from bug-fix dataset.	Generate mutants in seq2seq manner after learning from bug-fix dataset.	Randomly remove a code token and use an encoder-only transformer to predict the replacement.	Apply multiple mutation operators to the code randomly.

Complexity of code mutations.	Not limited.	Not limited.	Limited by the number or location of randomly removed code tokens.	Limited by the fixed set of mutation operators.
Consideration of code context during code mutations.	Good at interpreting far-apart code tokens relationships to determine the code statements to be mutated and the nature of code mutations.	Poor in interpreting the far-apart code tokens relationship to carry out code mutations.	Can interpret far-apart code token relationships but not mutating code in seq2seq manner.	Not considered.
Choice of mutation location in the code.	Determined by the transformer model based on the structure of the code that will be mutated.	Determined by the RNN model based on the structure of the code that will be mutated but limited by RNN weakness in interpreting far-apart code token relationships.	Random. Code token is removed randomly for the encoder-only transformer to predict a replacement token.	Mutation operators are randomly applied to the code.
Structure of machine learning model (if any).	Non-pre-trained or pre-trained transformers with encoder and decoder layers.	RNN with encoder and decoder layers.	Encoder-only transformers.	No machine learning model.

3. Methodology

After studied the background of MT and current situation of TMP, as well as the existing related works about TMP solutions, the three research questions (RQs) listed in section 1 above are formulated to guide this research. To provide the answers to the three RQs, the non-pre-trained transformer variants, and the pre-trained transformer variants are developed, trained, and fine-tuned using the bug-fix dataset by Tufano *et al.,* [8]. Then, the CHRF scores of the mutants that are produced by the transformer variants and the mutants produced by the state-of-the-art RNN, will be compared. Lastly, the generated mutants will be manually analysed to assess the nature of code mutations. The following subsections explain the development and training of the transformer model variants, as well as the steps to compare the performance of the machine learning models in generating mutants in seq2seq manner. Figure 2 shows the methodology flow of this study.

3.1 Development of Transformers Training, Fine-tuning, and Inference Code

The transformer models that will be involved in the experiment are non-pre-trained transformers with different number of encoder and decoder layers, transformers pre-trained with source code corpora which are CodeT5 and PLBART, as well as transformers pre-trained with natural language corpora which are Pegasus and Prophetnet. For all ML models, the training and testing dataset will use the same dataset as the state-of-the-art RNN. This is to ensure proper performance comparisons. The training dataset and testing dataset consist of pairs of fixed code and corresponding buggy code. All the code sequences in the dataset have been abstracted to ease the model training. For example, the variable, “studentNumber” is abstracted into “var_1” while the string, “operation completed successfully” is abstracted into “string_1”. Figure 3 shows the structure of transformer model. The input of the transformers is the code that will be mutated while the output is the mutated code.

By utilizing the PyTorch neural network module, the training code for non-pre-trained transformers is developed. In this case, five variants of the training code are written to compare the performance of non-pre-trained transformers with 1, 2, 3, 4, and 5 encoder and decoder layers in seq2seq code mutation. Besides, the inference code to test the non-pre-trained transformers is also developed. Listing 1 shows the algorithm of the training code of the non-pre-trained transformers, while Listing 2 shows the related inference code. The training process will be carried out for 500 epochs, as most models tend to converge by the 500th epochs, as shown in Figure 4.

Listing 1
Training code algorithm of the non-pre-trained transformers

Input: Training dataset containing pairs of fixed code and corresponding buggy code

Output: Trained transformer model with n encoder-decoder layers for seq2seq code mutation

Initialization:
Import the required libraries such as PyTorch

Initialize variables such as the number of epochs (500), batch size (16), and learning rate (3e-4)

Set n number of encoder and decoder layers

Load the training dataset containing fixed and buggy source code

Check for GPU availability and set the device accordingly

Initialize transformer from Pytorch nn.Transformer module with the parameters

Initialize the Adam optimizer and a learning rate scheduler

Define the loss function (Cross-Entropy Loss)

Tokenize dataset

Training loop:

For each epoch in the range [1, number of epochs]:

For each data batch:

Get input and target sequences

Perform forward pass, compute the loss, and update the model parameters

Record training loss

Save model checkpoint

Calculate the training loss

Save and plot the training loss

Listing 2
Inference code algorithm of the non-pre-trained transformers

Input: Trained transformer model with n encoder-decoder layers, and test dataset with code to be mutated and corresponding buggy code for comparison

Output: Mutants, training loss graph, mutant CHRF score

Initialization:
Import the required libraries such as PyTorch

Check for GPU availability and set the device accordingly

Load saved trained transformer model

Load test data that consists of fixed code to be mutated, and corresponding buggy code for comparison

Testing loop:

For each code to be mutated in test dataset

Load the code into the model to produce mutants

Calculate the CHRF score of the produced mutants based on corresponding buggy code in the test dataset

Calculate average CHRF score

Save mutants and CHRF score to the output file

As for the training of pre-trained transformers, the pre-trained models are loaded from Hugging Face, which is a hosting platform of ML models. Then, the code that fine-tunes the pre-trained transformers with the training dataset is developed. Besides, inference code is also developed to test the performance of non-pre-trained transformers in seq2seq code mutation. Listing 3 shows the algorithm of the fine-tuning code of the pre-trained transformers, while Listing 4 shows the related inference code. The training process will be carried out for 30 epochs, as most models tend to converge by the 30th epochs, as shown in Figure 5. Too many epochs may cause over-fitting.

Listing 3
Fine-tuning code algorithm of the pre-trained transformers

Input: Training dataset containing pairs of fixed and buggy sentences

Output: Fine-tuned pre-trained transformer model (CodeT5, PLBART, Pegasus, or Prophetnet) for mutant generation

Initialization:
Import the required libraries, including transformers, PyTorch, and other libraries

Initialize variables such as the number of epochs (30), batch size (16), and learning rate (1e-5)

Check for GPU availability and set the device accordingly

Load the training dataset containing fixed and buggy source code

Load a pre-trained model and tokenizer (CodeT5, PLBART, Pegasus, or Prophetnet)

Model Training:

Move the pre-trained model to the GPU if available

Define the optimizer (AdamW) and loss function (CrossEntropyLoss)

Training Loop:

For each epoch in the range [1, number of epochs]:

Shuffle the training examples to ensure randomness

For each data batch:

Tokenize the input and target sentences using the tokenizer

Create batch tensors for input and target code sequences as well as attention mask

Perform forward pass, compute the loss, and update the model parameters

Save the model checkpoint at the end of each epoch and record training loss

Calculate the training loss

Save and plot the training loss

Listing 4
Inference code algorithm of the pre-trained transformers

Input: Fine-tuned pre-trained transformer model (CodeT5, PLBART, Pegasus, or Prophetnet) & test dataset with code to be mutated and corresponding buggy code for comparison

Output: Mutants, training loss graph, mutant CHRF score

Initialization:
Import the required libraries, including transformers, PyTorch, and other libraries
Check for GPU availability and set the device accordingly

Load a pre-trained model and tokenizer

Load the checkpoint that was saved during training

Load test data that consists of fixed code to be mutated, and corresponding buggy code for comparison

Testing loop:

For each code to be mutated in test dataset

Load the code into the model to produce mutants

Calculate the CHRF score of the produced mutants based on corresponding buggy code in the test dataset

Calculate average CHRF score

Save mutants and CHRF score to the output file

3.2 Machine Learning (ML) Models Performance Comparison and Mutant Analysis

All the training code and fine-tuning code are run in a Python environment with access to P100 GPUs. Throughout the training or fine-tuning process, the training loss is recorded and plotted into a line graph for analysis. After training, the resulting ML models are loaded into the inference code, which mutates the code sequences in the test dataset and compare the mutated code with the real buggy code in the test dataset. The similarity between the produced mutants and the real buggy code is measured in terms of character n-gram F-score (CHRF).

Similar to BLEU score used in experiment by Tufano *et al.,* [8], CHRF score is also a metric that is used to evaluate the quality of machine translated sentences. It computes the similarity between the generated mutants with the target mutation pattern in the test dataset based on character n-grams. In this research, CHRF score is used instead of BLEU score because, according to the empirical analysis by Evtikhiev *et al.,* [35], CHRF score is closer to human assessment of machine translated sentence quality. CHRF score can be calculated using Eq. (2) where CHRP and CHRR represents precision and recall, respectively while β is the importance of recall with respect to precision [36].

$$CHRF = (1 + \beta)\frac{CHRP\ .\ CHRR}{\beta^2 . CHRP + CHRR} \qquad (2)$$

Lastly, for every ML models, 10 mutants are randomly selected so that their characteristics can be manually analysed. Due to space constraint, this paper will only illustrate a subset of the manually analysed mutants in Table 4 until Table 10. The full list of the manually analysed mutants can be founded in the online appendix [37]. This process is necessary because CHRF scores alone is not sufficient to gauge the quality of the produced mutants as it is not necessary for the produced mutants to be exactly the same as the buggy code in the test dataset. If a ML model can produce a sufficiently complex code mutation while maintaining syntax correctness, it can be considered satisfactory.

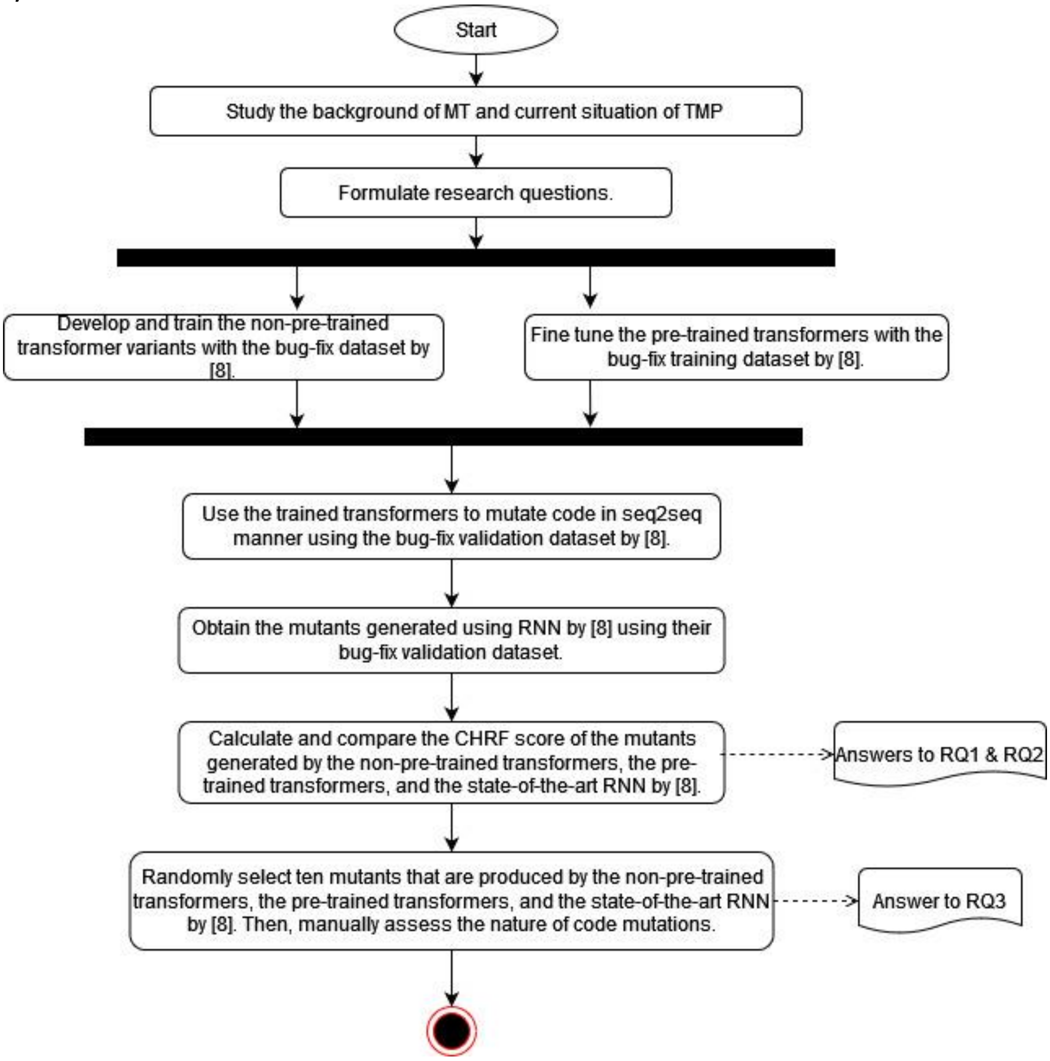

Fig. 2. Methodology flow of this study

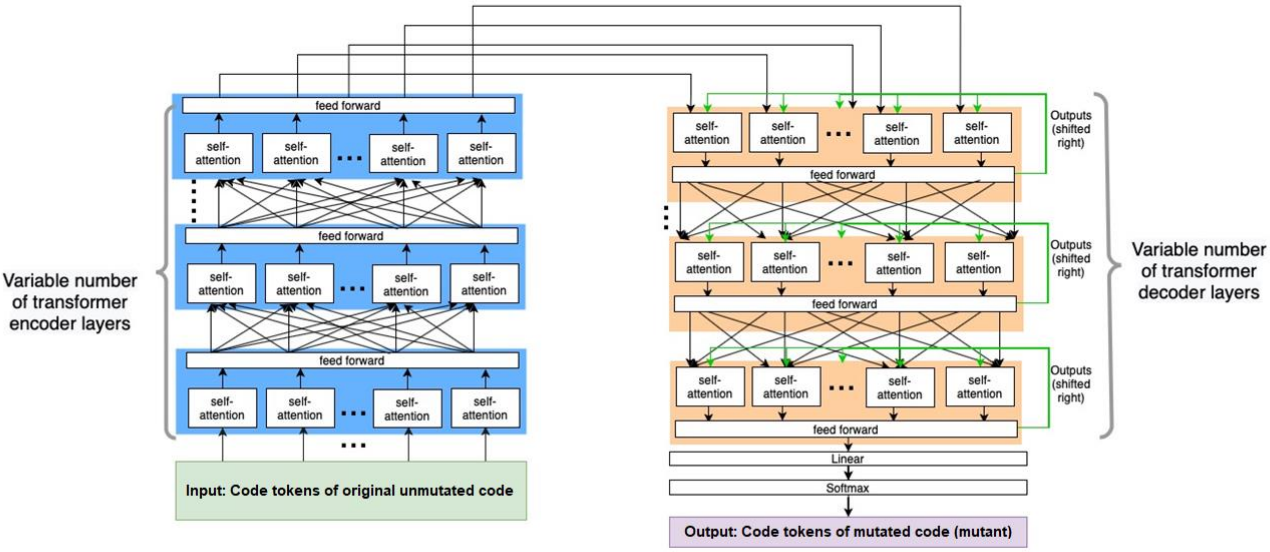

Fig. 3. Structure of transformer model

4. Results and Discussion

RQ1. Do the pre-trained transformers perform better than the non-pre-trained transformer and the state-of-the-art RNN in mutating code?

The average CHRF score of the mutants produced by the state-of-the-art RNN is 51.68. Before we can start judging whether the pre-trained transformers can perform better than the non-pre-trained ones in seq2seq code mutation, we need to first compare whether the mutants produced by the transformers is better than the state-of-the-art RNN. Based on the average CHRF score collected during the experiment as shown in Table 2 and Table 3, it is clear that transformers are capable of learning the bug patterns from the bug-fix dataset and use the knowledge to mutate the input code sequences. With its self-attention mechanism, positional encoding of input sequence tokens and the behaviour of interpreting input sequence tokens simultaneously, the transformers are able to preserve the structure of the code such as function definitions, appropriate braces, and function implementations, while injecting appropriate code mutations. All investigated transformer variants except Prophetnet, can generate mutants that have average CHRF scores of more than 70, and they are significantly higher than that of the state-of-the-art RNN (51.68). The high average CHRF scores indicate that the mutations made to the code in the test dataset are mostly resemble to the desired mutation patterns which are adapted from the real buggy code made by software developers.

CHRF scores alone is not sufficient to gauge the quality of the produced mutants as it is not necessary for the produced mutants to be exactly the same as the buggy code in the test dataset. During the manual analysis of the randomly selected mutants, we found out that the state-of-the-art RNN is more likely to produce mutants with syntax error compared to transformers. For example, as shown in Table 4, the mutants produced by the state-of-the-art RNN have “catch” scope immediately after “if” scope instead of “try” scope. The syntax error is likely due to the weakness of RNN in interpreting the dependencies between the code tokens that are far apart with each other. Meanwhile, the mutants produced by the state-of-the-art RNN also have simpler code mutation, unlike the transformer-generated mutants which have more complex code mutations such as changing conditional scope contents, addition of method calls, and wrapping method calls with conditional statements. The more complex code mutations are more capable to simulate real software faults for better MT.

Table 2
Final training loss, training time, and mutant CHRF score of non-pre-trained transformers

Number of encoder-decoder layers	1	2	3	4	5
Final loss	0.169029	0.045685	0.023704	0.017047	0.045188
Training time	9290	15504	21605	27897	34218
CHRF score	58.27	70.20	71.32	71.16	72.19

Table 3
Final training loss, training time, and mutant CHRF score of pre-trained transformers

Type	CodeT5	PLBart	Pegasus	Prophetnet
Final loss	0.001756	0.001189	0.009221	0.000838
Training Time	62547	86400	134220	183120
CHRF score	82.89	79.61	80.48	30.26

Table 4
RNN tend to produce mutants with syntax error

Example ID	Code Type	Code
eg8	Original code	public void METHOD_1 (TYPE_1 VAR_1 , int id) { for (TYPE_2 VAR_2 : new java.util.ArrayList (VAR_3)) { VAR_3 . remove (VAR_2) ; VAR_4 . METHOD_2 (VAR_2) ; } METHOD_3 ( ) ; METHOD_4 ( ) ; }
	RNN-mutated code	public void METHOD_1 (TYPE_1 VAR_1 , TYPE_1 VAR_2) { if (VAR_1 != null) { VAR_2 . METHOD_2 ( ) ; } catch (TYPE_1 VAR_3) { VAR_3 . METHOD_3 ( ) ; } } //useless mutant with syntax error
eg9	Original code	public void METHOD_1 (TYPE_1 status) { if (status . METHOD_2 ( )) { } else { TYPE_2 . METHOD_4 (VAR_1 , (STRING_1 + (status . METHOD_5 ( )))) ; METHOD_6 ( ) ; } }
	RNN-mutated code	public void METHOD_1 (TYPE_1 event) { super . METHOD_1 ( ) ; if ((VAR_1) != null) { METHOD_2 ( ) ; } catch (TYPE_1 VAR_1) { METHOD_3 ( ) ; } } //useless mutant with syntax error

Since it is now proven that transformers can perform better than the state-of-the-art RNN in seq2seq code mutation, we can now focus on investigating whether the pre-trained transformers can perform better than the non-pre-trained ones in seq2seq code mutation. Among the investigated pre-trained transformers, CodeT5 and PLBART are pre-trained with source code corpora while Pegasus and Prophetnet are pre-trained using natural language corpora. Except Prophetnet, all other pre-trained transformers can produce mutants that have average CHRF scores which are higher than that of the non-pre-trained transformers. Besides, as shown in Figure 4 and Figure 5, the pre-trained transformers also converge faster than the non-pre-trained transformers. This shows that the pre-initialized weights of the pre-trained transformers can contribute to improve the performance of the transformers in interpreting the input source code sequences. During pre-training, the transformers learn to interpret the syntax or structures of source code or natural languages. However, the results show that fine-tuning the pre-trained transformers for the downstream task of seq2seq code mutation, requires longer time than training non-pre-trained transformers for the same task.

During the manual analysis of the generated mutants, we found out that all the investigated pre-trained transformers except Prophetnet are more likely to produce mutants which are closer in nature to the real bugs. For example, as shown in Table 5, the difference between the buggy code and the corresponding fixed code from the test dataset is the presence of a conditional statement, the pre-trained PLBART and CodeT5 are able to mutate the fixed code and produce a mutant with removed conditional statement. The produced mutants are similar in nature with the buggy code from the test dataset. On the other hand, the pre-trained transformers except Prophetnet are also less likely to produce mutants with syntax errors compared to non-pre-trained transformers. For example, as shown in Table 6, the non-pre-trained transformer produced mutants with syntax errors. Meanwhile, none of the ten manually analysed mutants produced by Pegasus, PLBART and CodeT5 have syntax errors.

In short, the results justified that pre-trained transformers especially CodeT5 and PLBART which previously only tested with other seq2seq downstream tasks such as source code summarization and source code programming language translation, is also capable for seq2seq code mutation. Moreover, they also performed better than all investigated non-pre-trained transformers.

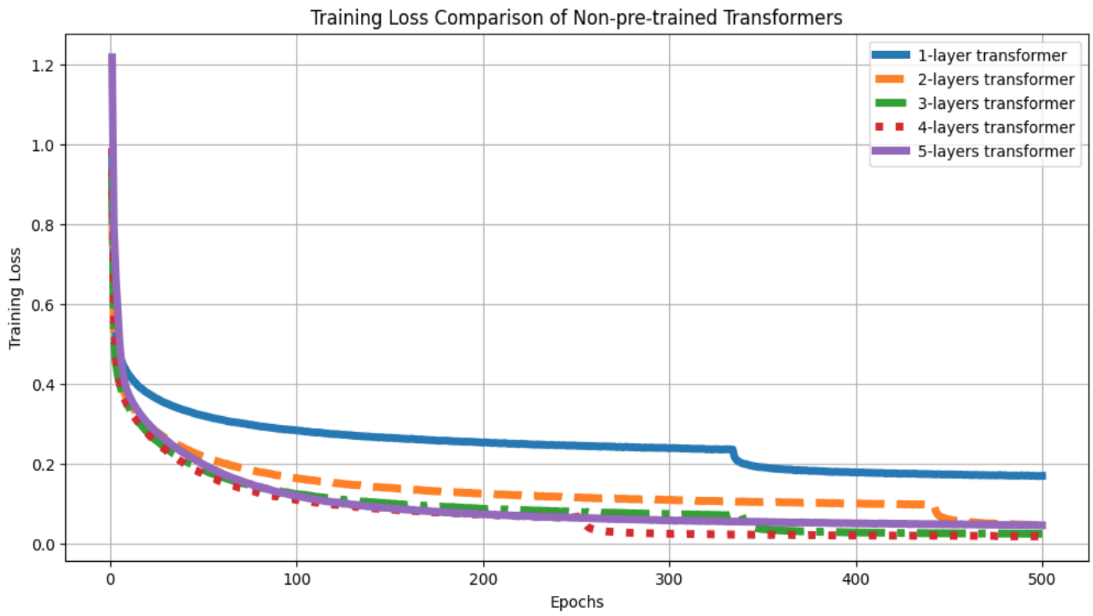

Fig. 4. Training loss of non-pre-trained transformers

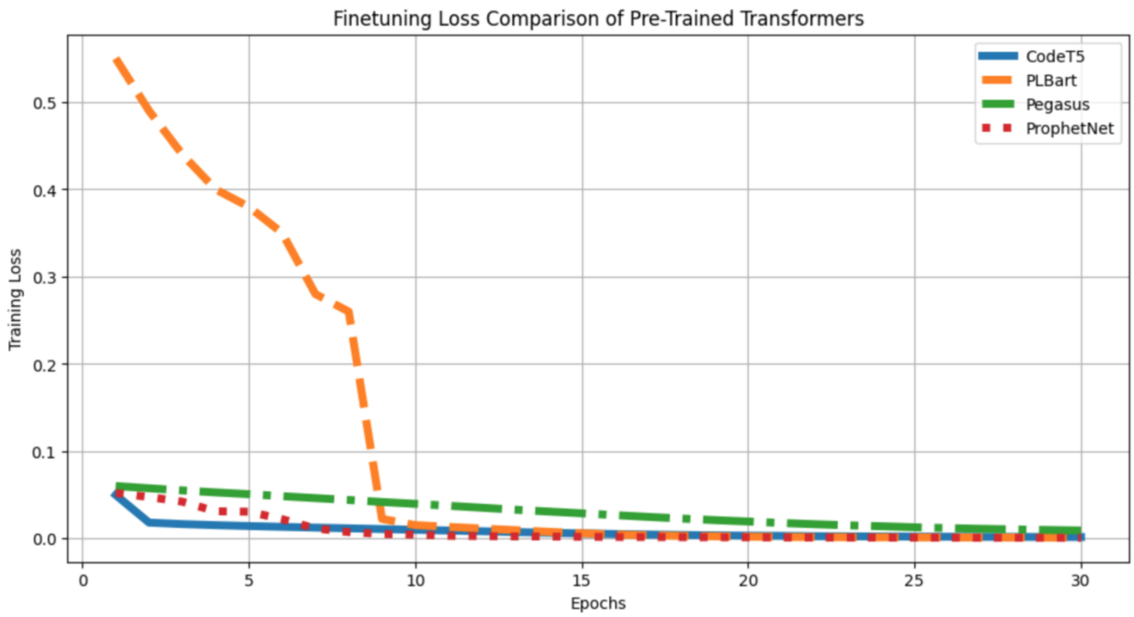

Fig. 5. Fine-tuning loss of pre-trained transformers

Table 5
CodeT5 and PLBART are able to produce mutants with the expected nature

Example ID	Code Type	Code
eg4	Original code	private static double METHOD_1 (java.util.ArrayList < java.lang.Double > p) { double sum = 0 ; for (double i : p) { if (i > 0) sum -= i * (TYPE_1 . METHOD_2 (i)) ; } return sum ; }
	Expected mutation	private static double METHOD_1 (java.util.ArrayList < java.lang.Double > p) { double sum = 0 ; for (double i : p) { sum -= i * (TYPE_1. METHOD_2 (i)) ; } return sum ; } //expectation: remove the enclosing "if"
	CodeT5-mutated code	private static double METHOD_1 (java.util.ArrayList < java.lang.Double > p) { double sum = 0 ; for (double i : p) { sum -= i * (TYPE_1. METHOD_2 (i)) ; } return sum ; } //produced mutant with expected nature
	PLBART-mutated code	private static double METHOD_1 (java.util.ArrayList < java.lang.Double > p) { double sum = 0 ; for (double i : p) { sum += TYPE_1. METHOD_2 (i) ; } return sum ; } //produced mutant with expected nature

Table 6
Unlike the pre-trained transformer, the non-pre-trained transformer may produce mutants with syntax error

Code Type	Example ID	Code
Original code	eg1	public void METHOD_1 ( ) { METHOD_3 ( ) ; if ((VAR_3) != null) VAR_3 . METHOD_1 (VAR_2) ; }
	eg4	private static double METHOD_1 (java.util.ArrayList < java.lang.Double > p) { double sum = 0 ; for (double i : p) { if (i > 0) sum -= i * (TYPE_1 . METHOD_2 (i)) ; } return sum ; }
Code by non-pre-trained 5-layer transformer	eg1	native void method_1 ( ) ; java.lang.system.out.println ((string_1 + (method_2 ( ))) ; method_3 ( ) ; if ((var_3) != null) var_3 . method_1 (var_2) ; } //useless mutant with syntax error
	eg4	static double method_1 (java.util.arraylist < java.lang.double > p) { double sum = 0 ; for (double i : p) { if (i < sum > 0) sum -= type_1 . method_2 (i) ; } return sum ; } //useless mutant with syntax error
CodeT5-mutated code	eg1	public void METHOD_1 ( ) { if ((VAR_1) >= (VAR_2)) { VAR_3. METHOD_1 (VAR_2) ; } METHOD_2 ( ) ; METHOD_3 ( ) ; } //mutant with no syntax error
	eg4	private static double METHOD_1 (java.util.ArrayList < java.lang.Double > p) { double sum = 0 ; for (double i : p) { sum -= i * (TYPE_1. METHOD_2 (i)) ; } return sum ; } //no syntax error
PLBART-mutated code	eg1	public void METHOD_1 ( ) { VAR_1 = METHOD_2 ( ) ; METHOD_3 ( ) ; if ((VAR_3)!= null) VAR_3. METHOD_1 (VAR_2) ; } //mutant with no syntax error
	eg4	private static double METHOD_1 (java.util.ArrayList < java.lang.Double > p) { double sum = 0 ; for (double i : p) { sum += TYPE_1. METHOD_2 (i) ; } return sum ; } //mutant with no syntax error
Pegasus-mutated code	eg1	public void METHOD_1 ( ) { if (((VAR_1)!= null) && ((VAR_2)!= null)) { METHOD_2 ( ) ; METHOD_3 ( ) ; if ((VAR_3)!= null) VAR_3. METHOD_1 (VAR_2) ; } } //mutant with no syntax
	eg4	public static double METHOD_1 (java.util.ArrayList < java.lang.Double > p) { double sum = 0 ; for (double i : p) { if (i < 0) sum -= i * (TYPE_1. METHOD_2 (x, 0)) ; } return sum ; } //mutant with no syntax

RQ2. Does the type of pre-training data and the pre-training method influence the transformers in the downstream task of seq2seq code mutation?

Among the four investigated transformers, CodeT5 and PLBART are the ones which are pre-trained with source code datasets. The average CHRF score that show CodeT5 is the best performing pre-trained transformer for seq2seq code mutation, justified that the pre-training process which involves code identifier tagging is indeed useful to improve the performance of transformers in interpreting source code structure. PLBART yields a slightly inferior CHRF score compared to CodeT5 while producing code mutation, because its pre-training process treats the source code datasets like normal natural language datasets.

Meanwhile, Pegasus and Prophetnet are pre-trained with natural language datasets. The results show Pegasus did surprisingly well in seq2seq code mutation while Prophetnet yields a very inferior performance. One assumption that can be made here is that Pegasus is pre-trained with text summarizations. Since Pegasus is pre-trained with text summarizations, one assumption that can be made is that the nature of summarizing text is quite similar to code mutation. So, fine-tuning Pegasus with the bugfix datasets can allow Pegasus to produce mutants in seq2seq manner as expected. On the other hand, Prophetnet is pre-trained to predict tokens and possible future tokens, which their information is then used as extra guidance to predict more future tokens for the output. One assumption that can be inferred from the poor Prophetnet performance is that, predicting possible future tokens may be suitable only for natural language which words at the latter part of the sentence can be more easily guessed based on a few words the early part of the sentence, as for our downstream task about seq2seq source code mutation, the appropriate code tokens in the latter part of the output sequence may not be accurately guessed based on the information of the code tokens at the early part of the code sequence. The poor performance of Prophetnet proven that, apart from pre-training datasets, the method of pre-training will also greatly influence the performance of the transformers in performing downstream tasks, which in this case, seq2seq code mutation.

RQ3. What are the characteristics of the mutants produced by the non-pre-trained transformers, the transformers pre-trained with source code corpora, the transformers pre-trained with natural language, and the state-of-the-art RNN model?

For every ML models, 10 mutants are randomly selected so that their characteristics can be manually analysed. The non-pre-trained transformers may sometimes generate mutants with syntax errors, but not as often as the state-of-the-art RNN. The weakness of RNN in interpreting long range token relationships is proven when the state-of-the-art RNN append “catch” scope after “if” scope instead of “try” scope as shown in Table 4. Moreover, RNN also have higher tendency in generating mutants with unnecessary extra brackets as shown in Table 7. Out of the ten randomly selected mutants, five of the mutants produced by the state-of-the-art RNN have syntax errors while only two of the mutants produced by the non-pre-trained transformers have syntax errors.

Even though the non-pre-trained transformers and the state-of-the-art RNN can produce mutants with correct syntax in some cases, the code modifications of the produced mutants are not as complex as those that are produced by the pre-trained transformers. For example, as shown in Table 7, the state-of-the-art RNN and the non-pre-trained transformer only mutate the return statements and function access level, respectively. PLBART, on the other hand, are able to add an extra conditional scope.

Unlike the non-pre-trained transformers, the code pre-trained transformers, CodeT5 and PLBART, are more capable in creating mutants that have higher potential to alter the program behaviour. For example, CodeT5 and PLBART can produce code modifications that involve adding extra method call, removing conditional statement, altering loop scope conditions, and wrapping existing lines with conditional scope as shown in Table 8.

Despite being a transformer pre-trained with natural language corpora, Pegasus can still produce code mutants that are less likely to have syntax errors. However, unlike mutants produced by CodeT5 and PLBART, the mutants produced by Pegasus tend to have less complex code modifications. For example, as shown in Table 9, Pegasus only mutate the return statement, while CodeT5 and PLBART is able to mutate the code by adding multiple conditional scopes with different return statements in each scope. However, in some cases, Pegasus may not conduct mutation to the input code sequence, especially when the input code sequence resembles buggy code that can be fixed with very minor code corrections. For example, as shown in Table 10, Pegasus does not made any changes to the initialization of the int variable, while both CodeT5 and PLBART are able to make a small change to the initialization of the int variable. This shows that, unlike Pegasus, CodeT5 and PLBART are capable to produce either mutants with a large degree of code modifications or mutants with minor code modification when necessary, depending on the nature of the input code sequences that will be mutated. Meanwhile, Prophetnet, which is also a transformer pre-trained with natural language corpora, has high tendency to produce mutants with syntax errors. Out of the ten manually analysed mutants, seven out of ten mutants produced by Prophetnet have syntax errors. This shows that not all transformers pre-trained with natural language are suitable to be fine-tuned for seq2seq code mutation, as the pre-training method will influence the performance of the transformers in carrying out the downstream tasks.

Table 7
Unlike RNN and non-pre-trained transformer, PLBART is able to produce a more complex mutation

Example ID	Code Type	Code
eg3	Original code	public static boolean equals (TYPE_1 VAR_1 , TYPE_1 VAR_2) { return VAR_1 . METHOD_1 (VAR_2) ; }
	RNN-mutated code	public boolean METHOD_1 (TYPE_1 VAR_1 , int VAR_2) { return ((TYPE_1 . METHOD_2 (VAR_1 . METHOD_2 ( )))) && ((((VAR_2 . METHOD_2 ( ))))) ; } //mutated return statement but it is a useless mutant due to syntax error
	Code by non-pre-trained 5-layer transformer	static double equals (type_1 var_1 , type_1 var_2) { return var_1 . method_1 (var_2) ; } //changed return type and function access level
	PLBART-mutated code	public static boolean equals (TYPE_1 VAR_1, TYPE_1 VAR_2) { if (false) throw new TYPE_2 (STRING_1) ; return VAR_1. METHOD_1 (VAR_2) ; } //extra "if" condition

Table 8
CodeT5 and PLBart can produce mutants that likely alter the system behaviour

Example ID	Code Type	Code
eg1	Original code	public void METHOD_1 ( ) { METHOD_3 ( ) ; if ((VAR_3) != null) VAR_3 . METHOD_1 (VAR_2) ; }
	Code by non-pre-trained 5-layer transformer	native void method_1 ( ) ; java.lang.system.out.println ((string_1 + (method_2 ( ))) ; method_3 ( ) ; if ((var_3) != null) var_3 . method_1 (var_2) ; } //useless mutant with syntax error
	CodeT5-mutated code	public void METHOD_1 ( ) { if ((VAR_1) >= (VAR_2)) { VAR_3. METHOD_1 (VAR_2) ; } METHOD_2 ( ) ; METHOD_3 ( ) ; } //extra method call and changed "if condition"
	PLBART-mutated code	public void METHOD_1 ( ) { VAR_1 = METHOD_2 ( ) ; METHOD_3 ( ) ; if ((VAR_3)!= null) VAR_3. METHOD_1 (VAR_2) ; } //extra method call added
eg4	Original code	private static double METHOD_1 (java.util.ArrayList < java.lang.Double > p) { double sum = 0 ; for (double i : p) { if (i > 0) sum -= i * (TYPE_1 . METHOD_2 (i)) ; } return sum ; }
	Code by non-pre-trained 5-layer transformer	static double method_1 (java.util.arraylist < java.lang.double > p) { double sum = 0 ; for (double i : p) { if (i < sum > 0) sum -= type_1 . method_2 (i) ; } return sum ; } //useless mutant with syntax error
	CodeT5-mutated code	private static double METHOD_1 (java.util.ArrayList < java.lang.Double > p) { double sum = 0 ; for (double i : p) { sum -= i * (TYPE_1. METHOD_2 (i)) ; } return sum ; } //line no longer enclosed by "if" scope
	PLBART-mutated code	private static double METHOD_1 (java.util.ArrayList < java.lang.Double > p) { double sum = 0 ; for (double i : p) { sum += TYPE_1. METHOD_2 (i) ; } return sum ; } //line no longer enclosed by "if" scope
eg8	Original code	public void METHOD_1 (TYPE_1 VAR_1 , int id) { for (TYPE_2 VAR_2 : new java.util.ArrayList (VAR_3)) { VAR_3 . remove (VAR_2) ; VAR_4 . METHOD_2 (VAR_2) ; } METHOD_3 ( ) ; METHOD_4 ( ) ; }
	Non-pre-trained 5-layer transformer-mutated code	void method_1 (type_1 var_1 , int id) { for (type_2 var_2 : var_3) { var_4 . method_2 (var_2) ; } method_3 ( ) ; method_4 ( ) ; }
	CodeT5-mutated code	public void METHOD_1 (TYPE_1 VAR_1, int id) { for (TYPE_2 VAR_2 : VAR_3) { VAR_3. remove (VAR_2) ; VAR_4. METHOD_2 (VAR_2) ; } METHOD_3 ( ) ; METHOD_4 ( ) ; } //changed "for" loop condition
	PLBart-mutated code	public void METHOD_1 (TYPE_1 VAR_1) { for (TYPE_2 VAR_2 : VAR_3) { VAR_3. remove (VAR_2) ; VAR_4. METHOD_2 (VAR_2) ; } METHOD_3 ( ) ; METHOD_4 ( ) ; } //changed "for" loop condition

eg10	Original code	public final void METHOD_1 (boolean VAR_1) { VAR_2 = VAR_1 ; METHOD_2 (VAR_1) ; }
	Non-pre-trained 5-layer transformer-mutated code	final void method_1 (boolean var_1) { var_2 = var_1 ; method_2 (var_1) ; }
	CodeT5-mutated code	public final void METHOD_1 (boolean VAR_1) { if (VAR_1) { VAR_2 = VAR_1 ; METHOD_2 (VAR_1) ; } else { VAR_2 = false ; } } //lines wrapped into conditionals
	PLBART-mutated code	public void METHOD_1 (boolean VAR_1) { VAR_2 = VAR_1 ; METHOD_2 (VAR_1) ; }

Table 9
CodeT5 and PLBart can make complex code mutations, unlike Pegasus

Example ID	Code Type	Code
eg5	Original code	public int method_1 ( ) { return var_1 . method_2 ( ) ; }
	Pegasus-mutated code	public int METHOD_1 ( ) { return VAR_1. METHOD_2 ( ). METHOD_3 ( ) ; } //mutation to return statement
	CodeT5-mutated code	public int METHOD_1 ( ) { if (VAR_1. isEmpty ( )) { return (VAR_2) ++ ; } else { return VAR_1. METHOD_2 ( ) ; } } //lines wrap into conditional statements
	PLBART-mutated code	public int METHOD_1 ( ) { if ((VAR_1)!= null) { return VAR_1. METHOD_2 ( ) ; } return - 1 ; } //lines wrap into conditional statements

Table 10
CodeT5 and PLBart can make small mutations when necessary, unlike Pegasus

Example ID	Code Type	Code
eg6	Original code	public int METHOD_1 ( ) { int VAR_1 = (value . METHOD_2 (INT_1)) + 1 ; return VAR_1 ; }
	Pegasus-mutated code	public int METHOD_1 ( ) { int VAR_1 = (value. METHOD_2 (INT_1)) + 1 ; return VAR_1 ; } //method content remains unchanged
	CodeT5-mutated code	public int METHOD_1 ( ) { int VAR_1 = (value. METHOD_2 (INT_1)) + INT_2 ; return VAR_1 ; } //small mutation made to method content
	PLBART-mutated code	public int METHOD_1 ( ) { value. METHOD_2 (INT_1) ; return VAR_1 ; } //small mutation made to method content

5. Conclusion and Future Work

In summary, this paper presented a comparison study between the non-pre-trained transformers, the transformers pre-trained with source code corpora, the transformers pre-trained with natural language corpora, and the state-of-the-art RNN model, in producing complex mutants that resemble realistic software faults made by programmers, which solves the TMP that lead to inaccurate mutation score. The results showed that the transformers pre-trained with source code yield a superior result with CodeT5 being the best achiever in terms of CHRF score and code mutation complexity. The source code which are related to this research paper are available at, https://github.com/LohZheungYik/TransformerMutation.

In the future, we will propose a mutation tool that utilizes the fine-tuned CodeT5 model. The tool will be used to create mutants of a software-under-test whose test suite inadequacies and ground-truth mutation score are known. Next, mutation analysis will be conducted to determine the mutation score of the test suite. The mutation score yielded by the mutants produced using the proposed tool, will be compared with the ground-truth mutation score. This is to confirm whether the transformer-generated mutants can produce accurate mutation score. Then, the tool will be integrated with the solutions that rectify the cost and equivalent mutant problem of MT, so that not only trivial mutant problem of MT is handled. The tool will be different from other existing MT solutions which only tackle one of the MT problems and give rise to other MT problems [18].

Acknowledgement
This work was funded by the Research Management Center (RMC), Universiti Teknologi Malaysia (UTM) and the Ministry of Higher Education Malaysia (MOHE) through the Fundamental Research Grant Scheme (FRGS) under vot number R.J130000.7828.5F677.

References

[1] Parsai, Ali, and Serge Demeyer. "Comparing mutation coverage against branch coverage in an industrial setting." *International Journal on Software Tools for Technology Transfer* 22, no. 4 (2020): 365-388. https://doi.org/10.1007/s10009-020-00567-y
[2] Rojas, José Miguel, and Gordon Fraser. "Code defenders: a mutation testing game." In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 162-167. IEEE, 2016. https://doi.org/10.1109/icstw.2016.43
[3] Hariri, Farah, August Shi, Vimuth Fernando, Suleman Mahmood, and Darko Marinov. "Comparing mutation testing at the levels of source code and compiler intermediate representation." In *2019 12th IEEE conference on software testing, validation and verification (ICST)*, pp. 114-124. IEEE, 2019. https://doi.org/10.1109/icst.2019.00021
[4] Petrovic, Goran, Marko Ivankovic, Bob Kurtz, Paul Ammann, and René Just. "An industrial application of mutation testing: Lessons, challenges, and research directions." In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 47-53. IEEE, 2018. https://doi.org/10.1109/ICSTW.2018.00027
[5] Nguyen, Quang Vu, and Lech Madeyski. "Addressing mutation testing problems by applying multi-objective optimization algorithms and higher order mutation." *Journal of Intelligent & Fuzzy Systems* 32, no. 2 (2017): 1173-1182. https://doi.org/10.3233/JIFS-169117
[6] Gopinath, Rahul, Carlos Jensen, and Alex Groce. "Mutations: How close are they to real faults?." *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 2014. https://doi.org/10.1109/ISSRE.2014.40
[7] Omar, Elmahdi, Sudipto Ghosh, and Darrell Whitley. "Subtle higher order mutants." *Information and Software Technology* 81 (2017): 3-18. https://doi.org/1r0.1016/j.infsof.2016.01.016
[8] Tufano, Michele, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. "Learning how to mutate source code from bug-fixes." In *2019 IEEE International conference on software maintenance and evolution (ICSME)*, p. 301-312. 2019. https://doi.org/10.1109/ICSME.2019.00046
[9] Wang, Liyang, Dantong Niu, Xinjie Zhao, Xiaoya Wang, Mengzhen Hao, and Huilian Che. "A comparative analysis of novel deep learning and ensemble learning models to predict the allergenicity of food proteins." *Foods 10*, no. 4 (2021): 809. https://doi.org/10.3390/foods10040809
[10] Stefenon, Stefano Frizzo, Laio Oriel Seman, Luiza Scapinello Aquino, and Leandro dos Santos Coelho. "Wavelet-Seq2Seq-LSTM with attention for time series forecasting of level of dams in hydroelectric power plants." *Energy* 274 (2023): 127350. https://doi.org/10.1016/j.energy.2023.127350
[11] Ghani, Miharaini Md, Wan Azani Wan Mustafa, Mohd Ekram Alhafis Hashim, Hafizul Fahri Hanafi, and Durratul Laquesha Shaiful Bakhtiar. "Impact of Generative AI on Communication Patterns in Social Media." *Journal of Advanced Research in Computing and Applications* 26, no. 1 (2022): 22-34. Accessed September 13, 2024.
[12] Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).
[13] Degiovanni, Renzo, and Mike Papadakis. "µbert: Mutation testing using pre-trained language models." In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, p. 160-169. 2022. https://doi.org/10.1109/ICSTW55395.2022.00039
[14] Wang, Yue, Weishi Wang, Shafiq Joty, and Steven CH Hoi. "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation." In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, p. 8696-8708. 2021. https://doi.org/10.18653/v1/2021.emnlp-main.685
[15] Ahmad, Wasi, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. "Unified Pre-training for Program Understanding and Generation." *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2021. https://doi.org/10.18653/v1/2021.naacl-main.211

[16] Ferretti, Claudio, and Martina Saletta. "Naturalness in Source Code Summarization. How Significant is it?." In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*, p. 125-134. 2023. https://doi.org/10.1109/ICPC58990.2023.00027
[17] Karmakar, Anjan, and Romain Robbes. "What do pre-trained code models know about code?." In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, p. 1332-1336. 2021. https://doi.org/10.1109/ASE51524.2021.9678927
[18] Yik, Loh Zheung, Wan Mohd Nasir bin Wan Kadir, and Noraini Binti Ibrahim. "A Systematic Literature Review on Solutions of Mutation Testing Problems." *In 2023 IEEE 8th International Conference On Software Engineering and Computer Systems (ICSECS)*, p. 64–71. 2023. https://doi.org/10.1109/ICSECS58457.2023.10256324
[19] Schwander, Florian, Rahul Gopinath, and Andreas Zeller. "Inducing subtle mutations with program repair." In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, p. 25-34. 2021. https://doi.org/10.1109/ICSTW52544.2021.00018
[20] Papadakis, Mike, Thierry Titcheu Chekam, and Yves Le Traon. "Mutant quality indicators." In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, p. 32-39. 2018. https://doi.org/10.1109/ICSTW.2018.00025
[21] Holling, Dominik, Sebastian Banescu, Marco Probst, Ana Petrovska, and Alexander Pretschner. "Nequivack: Assessing mutation score confidence." In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, p. 152-161. 2016. https://doi.org/10.1109/ICSTW.2016.29
[22] Durelli, Vinicius HS, Nilton M. De Souza, and Marcio E. Delamaro. "Are deletion mutants easier to identify manually?." In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, p. 149-158. 2017. https://doi.org/10.1109/ICSTW.2017.32
[23] Khanfir, Ahmed, Anil Koyuncu, Mike Papadakis, Maxime Cordy, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. "iBiR: Bug-report-driven fault injection." *ACM Transactions on Software Engineering and Methodology* 32, no. 2 (2023): 1-31. https://doi.org/10.1145/3542946
[24] Tan, Lili, Yunzhan Gong, and Yawen Wang. "A Model for Predicting Statement Mutation Scores." *Mathematics*, no. 9 (2019): 778. https://doi.org/10.3390/math7090778
[25] Vercacmmen, Sten, Markus Borg, and Serge Demeyer. "Validation of Mutation Testing in the Safety Critical Industry through a Pilot Study." In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, p. 334-343. 2023. https://doi.org/10.1109/ICSTW58534.2023.00064
[26] Van Nho, Do, Nguyen Quang Vu, and Nguyen Thanh Binh. "A solution for improving the effectiveness of higher order mutation testing." In *2019 IEEE-RIVF International Conference on Computing and Communication Technologies (RIVF)*, p. 1-5. 2019. https://doi.org/10.1109/RIVF.2019.8713650
[27] Wedyan, Fadi, Abdullah Al-Shishani, and Yaser Jararweh. "GaSubtle: A New Genetic Algorithm for Generating Subtle Higher-Order Mutants." *Information* 13, no. 7 (2022): 327. https://doi.org/10.3390/info13070327
[28] Lima, Jackson A. Prado, and Silvia R. Vergilio. "A multi-objective optimization approach for selection of second order mutant generation strategies." In *Proceedings of the 2nd Brazilian Symposium on Systematic and Automated Software Testing*, p. 1-10. 2017. https://doi.org/10.1145/3128473.3128479
[29] Wong, Chu-Pan, Jens Meinicke, Leo Chen, João P. Diniz, Christian Kästner, and Eduardo Figueiredo. "Efficiently finding higher-order mutants." In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, p. 1165-1177. 2020. https://doi.org/10.1145/3368089.3409713
[30] Chen, Yaocong, Mingyuan Fan, Shahbaz Gul Hassan, Jiawei Lv, Bing Zhou, Wenting Fan, Jingbin Li et al. "Waterfowl breeding environment humidity prediction based on the SRU-based sequence to sequence model." *Computers and Electronics in Agriculture* 201 (2022): 107271. https://doi.org/10.1016/j.compag.2022.107271
[31] Han, Shi-Yuan, Qiang Zhao, Qi-Wei Sun, Jin Zhou, and Yue-Hui Chen. "EnGS-DGR: Traffic Flow Forecasting with Indefinite Forecasting Interval by Ensemble GCN, Seq2Seq, and Dynamic Graph Reconfiguration." *Applied Sciences* 12, no. 6 (2022): 2890. https://doi.org/10.3390/app12062890
[32] Zhang, Qihang, and Bin Wu. "Software defect prediction via transformer." In *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, p. 874-879. 2020. https://doi.org/10.1109/ITNEC48623.2020.9084745
[33] Zhang, Jingqing, Yao Zhao, Mohammad Saleh, and Peter Liu. "Pegasus: Pre-training with extracted gap-sentences for abstractive summarization." In *International Conference on Machine Learning*, p. 11328-11339. 2020. https://dl.acm.org/doi/abs/10.5555/3524938.3525989
[34] Qi, Weizhen, Yu Yan, Yeyun Gong, Dayiheng Liu, Nan Duan, Jiusheng Chen, Ruofei Zhang, and Ming Zhou. "ProphetNet: Predicting Future N-gram for Sequence-to-Sequence Pre-training." In *Findings of the Association for Computational Linguistics: EMNLP 2020*, p. 2401-2410. 2020. https://doi.org/10.18653/v1/2020.findings-emnlp.217

[35] Evtikhiev, Mikhail, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. "Out of the bleu: how should we assess quality of the code generation models?." *Journal of Systems and Software* 203 (2023). https://doi.org/10.1016/j.jss.2023.111741

[36] Popović, Maja. "chrF: character n-gram F-score for automatic MT evaluation." In *Proceedings of the tenth workshop on statistical machine translation*, p. 392-398. 2015. https://doi.org/10.18653/v1/W15-3049

[37] Yik, Loh Zheung, Wan Mohd Nasir bin Wan Kadir, and Noraini Binti Ibrahim. *Appendix to Paper: A Comparative Evaluation of Transformers in Seq2Seq Code Mutation: Non-pre-trained vs. Pre-trained Variants*, Google Drive link, accessed September 14, 2024.