On Multi-Modal Learning of Editing Source Code

Saikat Chakraborty
Department of Computer Science
Columbia University
New York, NY, USA
saikatc@cs.columbia.edu

Baishakhi Ray
Department of Computer Science
Columbia University
New York, NY, USA
rayb@cs.columbia.edu

***Abstract*—In recent years, Neural Machine Translator (NMT) has shown promise in automatically editing source code. Typical NMT based code editor only considers the code that needs to be changed as input and suggests developers with a ranked list of patched code to choose from - where the correct one may not always be at the top of the list. While NMT based code editing systems generate a broad spectrum of plausible patches, the correct one depends on the developers' requirement and often on the context where the patch is applied. Thus, if developers provide some hints, using natural language, or providing patch context, NMT models can benefit from them.**

As a proof of concept, in this research, we leverage three modalities of information: edit location, edit code context, commit messages (as a proxy of developers' hint in natural language) to automatically generate edits with NMT models. To that end, we build MODIT, a multi-modal NMT based code editing engine. With in-depth investigation and analysis, we show that developers' hint as an input modality can narrow the search space for patches and outperform state-of-the-art models to generate correctly patched code in top-1 position.

***Index Terms*—Source Code Edit, Neural Networks, Automated Programming, Neural Machine Translator, Pretraining, Transformers**

I. INTRODUCTION

Programmers often develop software incrementally, adding gradual changes to the source code. In a continuous software development environment, programmers modify their source code for various reasons, including adding additional functionality, fixing bugs, refactoring, etc. It turns out that many of these changes follow repetitive edit patterns [1]–[3] resulting in a surge of research effort to automatically generate code-changes learned from past examples [2]–[6].

In particular, Neural Machine Translation (NMT) models have been successful in learning automatic code changes [5]–[11]. At the core, these models contain an encoder and a decoder — the encoder encodes the code that needs to be edited, and the decoder sequentially generates the edited code. Such NMT models are trained with a large corpus of previous edits to learn generic code change patterns. In the inference time, given a code fragment that needs to be edited, a trained NMT model should automatically generate the corresponding edited code.

However, learning such generic code changes is challenging. A programmer may change an identical piece of code in different ways in two different contexts, both can potentially be correct patches (see Figure 1). For example, an identical code fragment `pictures = new ArrayList<>()` was changed in two different ways: `pictures = new HashSet<>();` and `pictures = new LinkedList<>()` in two different code contexts. Without knowing the developers' intention and the edit context, the automated code editing tools have no way to predict the most intended patches. For instance, in the above example, `LinkedList` was used to fix a sublist-related problem. Once such an intention is known, it is easy to choose a LinkedList-related patch from the alternate options. Thus, such an additional modality of information can reinforce the performance of automated code-editing tools.

```
//Guidance: use LinkedList and fix sublist problem ...
public void addPicture (String picture){
    if ((pictures) == null) {
-       pictures = new ArrayList<>();
+       pictures = new LinkedList<>(); //correct patch
+       pictures = new HashSet<>(); //plausible  patch
    }
    pictures.add(picture);
}
```

Fig. 1: Example of an identical code (marked in red) changed in two different ways (green and blue) in two different contexts, where both can be correct patches. However, based on developers' guidance (top line) to fix a list related problem, green is the correct patch in this context.

```
1  // Guidance: fix problem which occurred when
2  // the resulting json is empty ...
3
4  private String generateResultingJsonString(
5          char wrappingQuote, Map<String, Object>jsonMap){
6      JsonObject jsonObject = new JSONObject(jsonMap);
7      String newJson = jsonObject.toJSONString(LT_COMPRESS);
8      if (
9  -       newJson.charAt(1) != wrappingQuote
10 +       !jsonObject.isEmpty() &&
11 +       (newJson.charAt(1) != wrappingQuote)
12     ){
13     return replaceUnescaped(
14        newJson, newJson.charAt(1), qrappingQuote);
15     }
16     return newJson;
17 }
```

Guidance → Context

Fig. 2: A motivating example. The guidance provides a brief summary of what needs to be changes. The underlined tokens are directly copied from guidance and context into the patched code.

In fact, given just a piece of code without any additional information, it is perhaps unlikely that even a human developer can comprehend how to change it. Consider another real-life

example shown in Figure 2. If a programmer *only* considers the edited expression in line 9, it is difficult to decide how to modify it. However, with additional information modalities – *i.e.,* the guidance (line 1,2) and the context (the whole method before the patch), the correct patch often becomes evident to the programmer since the guidance effectively summarises how to change the code and the context provides necessary ingredients for generating a concretely patched code. We hypothesize that such multi-modal information could be beneficial to an automated code-editing tool. To that end, we design MODIT, a multi-modal code editing engine that is based on three information modalities: (i) the code fragment that needs to be edited, (ii) developers' intention written in natural language, and (iii) explicitly given edit context.

In particular, MODIT is based on a transformer-based [12] NMT model. As input, MODIT takes the code that needs to be edited (*e.g.,* the lines that need to be patched), additional guidance describing developers' intent, and the context of the edits that are explicitly identified by the developer (*e.g.,* the surrounding method body, or surrounding lines of code, etc.). Note that previous works [6], [11] also provided context and the edit location while generating edits; however, they are fed together to the model as a unified code element. Thus, the model had the burden of identifying the edit location and then generating the patch. In contrast, isolating the context from the edit location and feeding them to the model as different modalities provides MODIT with additional information about the edits.

Curating developers' intent for a large number of edits that can train the model is non-trivial. As a proof of concept, we leverage the commit messages associated with the edits to simulate developers' intent automatically. We acknowledge that commit messages could be noisy and may not always reflect the change summary [13]. Nonetheless, our extensive empirical result shows that, even with such noisy guidance, MODIT performs better in generating correctly edited code.

Being a model that encodes and generates source code, MODIT needs to both clearly understand and correctly generate programming languages (PL). While several previous approaches [6], [14] designed sophisticated tree/grammar-based models to embed the knowledge of PL into the model, the most recent transformer-based approaches [15]–[17] showed considerable promise with pre-training with a large volume of source code. Since these models are pre-trained with billions of source code written by actual developers, and transformers are known to learn distant dependencies between the nodes, these models can learn about code structures during the pre-training step. Among such pre-trained models, PLBART [17] learns jointly to understand and generate source code and showed much promise in generative tasks. Thus, we chose PLBART as the starting point to train MODIT, i.e., we initialize MODIT's model with learned parameters from PLBART.

We evaluate MODIT on two different datasets ($B2F_s$, and $B2F_m$) proposed by Tufano *et al.* [8] consisting of an extensive collection of bug-fix commits from GitHub. Our empirical investigation shows that a summary of the change written in natural language as additional guidance from the developer improves MODIT's performance by narrowing down the search space for change patterns. The code-edit context, presented as a separate information modality, helps MODIT to generate edited code correctly by providing necessary code ingredients (*e.g.,* variable names, method names, *etc.*). MODIT generates $\sim$3.5 times more correct patches than CODIT showing that MODIT is robust enough to learn PL syntax implicitly. Furthermore, MODIT generates two times as many correct patches as a large transformer model could generate.

Additionally, our empirical investigation reveals that when we use one encoder to encode all information modalities rather than learning from individual modalities separately, the model learns representation based on inter-modality reasoning. In contrast, a dedicated encoder for each individual modality only learns intra-modality reasoning. Our experiment shows that a multi-modal/single-encoder model outperforms multi-modal/multi-encoder model by up to 46.5%.

We summarize our main contributions in this paper as follows.

- We propose MODIT– a novel multi-modal NMT-based tool for automatic code editing. Our extensive empirical evaluation shows that Automatic Code Editing can be vastly improved with additional information modalities like code context and developer guidance.
- We empirically investigate different design choices for MODIT. We provide a summary of the lessons that we learned in our experiments. We believe such lessons are valuable for guiding future research.
- We prototype and build MODIT and open-source all our code, data in https://git.io/JOudU.

II. BACKGROUND

A. Neural Machine Translation

Neural Machine Translation(NMT) [18] is a very well studied field, which has been very successful in translating a sentence from one language to another. At a very high level, input to an NMT model is a sentence ($X = x_1, x_2, ..., x_n$), which is usually a sequence of tokens (x_i), and the output is also a sentence ($Y = y_1, y_2, ..., y_m$) – sequence of tokens (y_i). While learning to translate from X to Y, NMT models learn conditional probability distribution $P(Y|X)$. Such probability distributions are learned *w.r.t.* model parameters θ, where model training process optimizes θ in such a way that maximizes the expected probability distribution of a dataset. An NMT model usually contains an encoder and a decoder. The encoder processes, understands, and generates vector representations of the input sentence. The decoder starts after the encoder and sequentially generates the target sentence by reasoning about the encoder-generated input representation. While sequentially generating the target sentence, the decoder usually performs different heuristic searches (for instance, beam search) to balance exploration and exploitation.

In recent few years, Software Engineering has seen a wide spectrum of adaptation of NMT. Some prominent application

of NMT is SE include Program Synthesis [19], Code summarization [20], [21], Edit summarization [13], Code Edit Generation [5], [6], [8], Automatic Program Repair [9]–[11], etc. These research efforts capitalize on NMTs' capability to understand and generate complex patterns and establish NMT as a viable tool for SE-related tasks.

B. Transformer Model for Sequence Processing

Transformer [12] model revolutionized sequence processing with attention mechanism. Unlike the traditional RNN-based model where input tokens are processed sequentially, the transformer assumes soft-dependency between each pair of tokens in a sequence. Such dependency weights are learned in the form of attention weights based on the task of the transformer. While learning the representation of a token, the transformer learns to attend to all the input tokens. From a conceptual point of view, the transformer converts a sequence to a complete graph[1], where each node is a token. The weights of the edges are attention weights between tokens which are learned based on the task of the transformer. The transformer encodes each token's position in the sequence (positional encoding) as part of the input. In such a way, the transformer learns long-range dependency. Since its inception, the transformer is very successful in different NLP understanding and generation tasks. Transformers' ability of reasoning about long-range dependency is proved useful for several source code processing task including code completions [22], code generation [23], code summarization [21].

C. Transfer Learning for Source Code

In recent few years, Transfer learning shows promise for a wide variety of SE tasks. Such transfer learning aims at learning task agnostic representation of source code and reuse such knowledge for different tasks. One way to learn such task agnostic representation of input is pre-training a model with a large collection of source code. The learning objective of such pre-training is often understanding the code or generating the correct code. A pre-trained model is expected to embed the knowledge about source code through its parameters. Such pre-trained models are later fine-tuned for task-specific objectives. CuBERT [24], CodeBERT [15], GraphCodeBERT [16] are all transformer-based encoder models which are pre-trained to understand code. Such models are primarily trained using Masked Language Model [25], replaced token prediction [15], semantic link prediction [16], etc.

For code generation, CodeGPT [9], [26] pre-trains a transformer-based model to generate general-purpose code sequentially. More recently, PLBART [17] pre-trained transformer-based model jointly for understanding and generating code with denoising auto-encoding [27]. PLBART consists of an encoder and a decoder. The encoder is presented with slight noise (for instance, token replacement) induced code, and the decoder is expected to generate noise-free code. Since code editing task requires both the understanding of code and code generation, we chose PLBART as the base model for MODIT.

[1]https://en.wikipedia.org/wiki/Complete_graph

III. MODIT

Figure 3 shows an overview of MODIT's working procedure. MODIT is a multi-layer encoder-decoder based model consisting of a Transformer-based encoder and a Transformer-based decoder. Both the encoder and decoder consist of 6 layers. MODIT works on three different modalities of information: (i) Code that needs to be edited (e_p), (ii) natural language guidance from the developer ($\mathcal{G}$), and (iii) the context code where the patch is applied (C). We acknowledge that e_p is essentially a substring of C. However, by explicitly extracting and presenting e_p to MODIT, we provide MODIT with additional information about the change location. Thus, despite being a part of the context, we consider e_p a separate modality. Nevertheless, MODIT consists of three steps. First, the pre-processing step processes and tokenizes these input modalities (§III-A). Then the encoder in MODIT encodes the processed input, and the decoder sequentially generates the patched code as a sequence of tokens (§III-B). At final step, MODIT post-processes the decoder generated output and prepares the edited code (§III-C).

A. Pre-processing

Input Consolidation. In the pre-processing step, MODIT generates consolidated multi-modal input (X) from the three input modalities (*i.e.,* e_p, $\mathcal{G}$, and C). MODIT combines these input modalities as a sequence separated by a special `<s>` token *i.e.,* $X = e_p$ `<s>` $\mathcal{G}$ `<s>` C. In the example shown in Figure 2, e_p is `newJson.charAt(1))!= wrappingQuote` , $\mathcal{G}$ is `fix problem which occurred when the resulting json is empty`, and C is the whole function before the edit (see *Input Modalities* in Figure 3). MODIT generates a consolidates multi-modal input sequence as `newJson.charAt(1)) ... <s> fix problem which occurred ... <s> private String ... }`.

Tokenization. MODIT uses sentence-piece tokenizer [28]. Sentence-piece tokenizer divides every token into sequence of subtokens. Such subword tokenization is similar to previously used byte-pair encoding in automatic code editing literature [9], [29]. We use PLBART [17]'s sentence-piece tokenizer which is trained on billions of code from GitHub. After tokenizing the consolidated input X from Figure 2, we get `new Json . char At ( 1 ) ... <s> fix problem which oc cur red ... <s> private String ... _}`.

B. Encoder-Decoder Model

The input to MODIT's encoder-decoder model is a sequence of subtokens generated in the previous step.

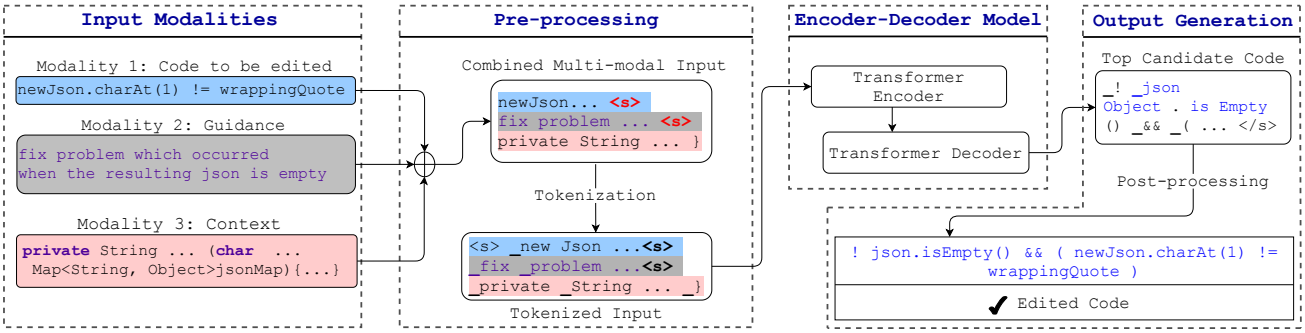


Fig. 3: Overview of MODIT pipeline

Transformer Encoder. Given an input sequence $X = x_1, x_1, ..., x_n$, the encoder learns the representation of every token at layer l as $R^e_l(x_i)$ using self-attention computed as

$$R^e_l(x_i) = \sum_{j=i}^{n} a_{i,j} * R^e_{l-1}(x_j) \quad (1)$$

Where $R^e_{l-1}(x_j)$ is the representation of subtoken x_j as generated by layer $l-1$, and $a_{i,j}$ is the attention weight of subtoken x_i to x_j. Such attention weights are learned by multi-head attention [12]. Final layer generated representation (*i.e.,* $R^e_6(x_i)$) is the final representation for every subtoken x_i in the input. Note that, the encoder learns the representation of Equation (1) of a subtoken, using all subtokens in the sequence. Thus the learned representation of every subtoken contains information about the whole input sequence. Since we encode all the information modalities in one sequence, the learned representation of every subtoken encodes information about other modalities.

Transformer Decoder. The decoder in MODIT is a transformer-based sequential left-to-right decoder consisting of 6 layers. It sequentially generates one subtoken at a time using previously generated subtokens and the final representation ($R^e_l(x_i)$) from the encoder. The decoder contains two modules – (i) self-attention, and (ii) cross-attention. The self-attention layer work similar to the self-attention in the encoder. First, with self attention, decoder generates representation $R^dl(y_i)$ of last generated token y_i with self attention on all previously generated tokens $(y_1, y_2, ..., y_i)$. This self attention follows same mechanism described in Equation (1). After learning decoder representation by self attention, decoder applies attention of encoder generated input representation using the following equation,

$$\mathcal{D}_l(y_i) = \sum_{j=i}^{n} \alpha^l_{i,j} * R^e_6(x_j) \quad (2)$$

Where $\alpha^l_{i,j} = softmax\left(dot\left(R^e_6\left(x_j\right), R^dl\left(y_i\right)\right)\right)$ is the attention weight between output subtoken y_i to input subtoken x_j. The softmax generates an attention probability distribution over the length of input tokens. Finally the decoder learned representation, $D_l(y_i)$ is projected to the vocabulary to predict maximally likely subtoken from the vocabulary as next token.

In summary, the encoder learns representation of every subtokens in the input using all input subtoken, essentially encoding the whole input information in every input subtoken representation. The decoder's self-attention mechanism allows the decoder to attend to all previously generated subtokens allowing the decoder decide on generating correct token at correct place. The cross-attention allows the decoder to attend to encoded representation - implicitly letting the model decide where to copy from the input where to choose from new tokens in the vocabulary. We initialize the end-to-end encoder-decoder in MODIT using pre-trained weights of PLBART [17].

C. Output Generation

The decoder in MODIT continue predicting subtoken until it predicts the end of sequence `</s>` token. During inference, MODIT uses beam search to generate sequence of subtokens. Once the decoder finishes, MODIT post-processes the top ranked sequence in the beam search. First, MODIT removes the end of sequence `</s>` token. It then detokenizes the subtokens sequence to code token sequence. In this step, MODIT merges generated subtokens that are fragments of a code token into one code token. For the example shown in Figure 2, MODIT generates the subtoken sequence `! json . is Empty () && ( new Json . char At ( 1 ) != wrap ping Quote ) </s>`. After detokenization, MODIT generates `! json.isEmpty()&& ( newJson.charAt(1)!= wrappingQuote ) .`

IV. EXPERIMENTAL DESIGN

A. Dataset

TABLE I: Statistics of the datasets studied.

Dataset	Avg. Tokens	Avg. Change Size*	Avg. tokens in Guidance	# examples		
				Train	Valid	Test
$B2F_s$	32.27	7.39	11.55	46628	5828	5831
$B2F_m$	74.65	8.83	11.48	53324	6542	6538

* Change size measured as token edit distance.

To prove our concept of MODIT, we experiment on two different datasets (*i.e.,* $B2F_s$, and $B2F_m$) proposed by Tufano *et al.* [8]. In these two datasets, they collected large collections of bug-fix code changes along with commit messages from Java projects in GitHub. Each example in these datasets contains the java method before the change (C_p), the method after the change (C_n), and the commit message for

the change. There are some examples (< 100) with corrupted bytes in the commit message, which we could not process. We excluded such examples from the dataset. Table I shows statistics of the two datasets we used in this paper. $B2F_s$ contains smaller methods with maximum token length 50, and $B2F_m$ contains bigger methods with up to 100 tokens in length. The average size of the change (edit distance) is 7.39, and 8.83 respectively, in $B2F_s$ and $B2F_m$.

B. Data Preparation

For the datasets described in Section IV-A, we extract the input modalities and the expected output to train MODIT. For every method pair (*i.e.,* before edit - C_p, after edit - C_n) in those dataset, we use GumTree [30] to extract a sequence of tree edit locations. We identify the root of the smallest subtree of C_p's AST that encompasses all the edit operations. We call the code fragment corresponding to that subtree as *code to be edited*(e_p) and used as MODIT's first modality. Similarly, we extract the code corresponding to the smallest subtree encompassing all the edit operations from C_n and use that as *code after edit*(e_n). We use the commit message associated with the function pair as MODIT's second modality, guidance($\mathcal{G}$). Finally, we use the full method before edit (C_p) as MODIT's third modality, context(C).

C. Training

After combining every example in the datasets in MODIT's input (e_p, $\mathcal{G}$, C) and expected output (e_n), we use this combined dataset to train MODIT. For training MODIT, we use Label Smoothed Cross Entropy [31] as loss function. We use Adam optimizer, with a learning rate of $5e^{-5}$. We train MODIT for 30 epochs, after every epoch, we run beam search inference on the validation dataset. We stop training if the validation performance does not improve for five consecutive validations.

D. Evaluation Metric

We use the top-1 accuracy as the evaluation metric throughout the paper. For proof-of-concept, we evaluate all techniques with beam size 5. When the generated patched code matches *exactly* with the expected patched code e_n, it is correct, incorrect otherwise. Note that this is the most stringent metric for evaluation. Previous approaches [6], [10], [11] talked about filtering out infeasible patches from a ranked list of top k patches using test cases. However, we conjecture that such test cases may not always be available for general purpose code edits. Thus, we only compare top-1 accuracy.

E. Research Questions

MODIT contains several design components: (i) use of multimodal information, (ii) use of transformer and initializing it with the pre-trained model, and (iii) use of end-to-end encoder-decoder (using PLBART) to generate patches instead of separately using pre-trained encoder or pre-trained decoder, as used by previous tools. First, we are interested in evaluating MODIT *w.r.t.* state-of-the-art methods. In particular, we evaluate how these three design choices effect MODIT's performance. So, we start with investigating,

RQ1. How accurately does MODIT generate edited code *w.r.t.* other techniques?

MODIT uses three input modalities. Our next evaluation target is how these individual modalities effect MODIT's performance? Thus we ask,

RQ2. What are the contribution of different input modalities in MODIT 's performance?

Finally, recall from Section III-A, MODIT proposes to encode all the input modalities as a sequence and use one encoder for the consolidated multi-modal input. An alternative to this encoding mechanism is to encode individual input modality with dedicated input encoder. Our next evaluation aims at finding out the best strategy to encode input modalities. Hence, we investigate,

RQ3. What is the best strategy to encode multiple input modalities?

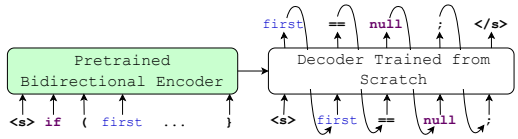
V. EMPIRICAL RESULTS

In our first research question, we evaluate MODIT's performance *w.r.t.* other techniques and the effect of MODIT's design components.

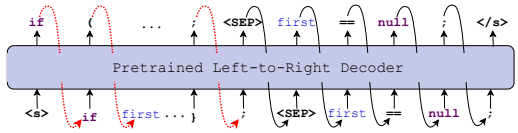
RQ1. How accurately does MODIT generate edited code *w.r.t.* other techniques?

Experimental Setup. We carefully chose the baselines to understand the contribution from different design choices of MODIT. We evaluated our model in two experimental settings. First, we train different baseline models where the full model is trained from scratch. In this setting, the first baseline we consider is an LSTM with attention [18] NMT model. Various existing code patching approaches [5], [7], [8], [11] used such settings. Second baseline is Transformer [12] based $\mathcal{S}2\mathcal{S}$ model. We consider two different-sized transformers. This enables us to contrast effect of model size in code-editing performance. The *Transformer-base* model consists of six encoder layers and six decoder layers. The *Transformer-base* model's architecture is the same as MODIT's architecture. Furthermore, we consider another transformer with a much larger architecture. *Transformer-large* contains twelve encoder layers and twelve decoder layers with three times as many learnable parameters as the *Transformer-base* model. The final baseline in this group is CODIT, which is a tree-based model. Comparison *w.r.t.* CODIT allows us to contrast externally given syntax information (in the form of CFG) and learned syntax by transformers (*i.e.,* MODIT). We use all three input modalities (see Figure 3 for example) as input to the LSTM and Transformer. Using auxiliary modalities is non-trivial with CODIT since the input to CODIT must be a syntax-tree. Thus, we use uni-modal input (e_p) with CODIT.

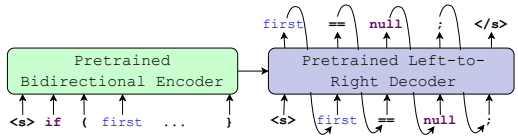
In the second setting, we consider different pre-trained models, which we used to fine-tune for patch generation. Figure 4 shows schematic diagrams of the pre-trained models we compared in this evaluation. First two models we considered are CodeBERT [15], and GraphCodeBERT [16]. Both



(a) CodeBERT — Consist of bidirectional pretrained encoder and a decoder trained from scratch.



(b) CodeGPT — One pretrained single decoder processes the input and output sequentially from left to right.



(c) PLBART — Consist of pretrained bidirectional encoder and pretrained left to right decoder.

Fig. 4: Schematic diagram of the three types of pre-trained models. used to evaluate MODIT.

of these models are pretrained encoders primarily trained to understand code. To use these for the patching task, we add a six-layered transformer-based decoder along with the encoder. The decoder is trained from scratch (see Figure 4a). Another pre-trained baseline is CodeGPT [26]. GPT is a single left-to-right decoder model primarily pre-trained to generate code. For the code editing task, a special token `<SEP>` combines the input and the output as a sequence separated. Jiang *et al.* [9] showed the effectiveness of GPT for the source code patching task (see Figure 4b). In contrast to these pre-trained models, MODIT uses PLBART, an end-to-end encoder-decoder model trained to understand and generate code simultaneously (see Figure 4c). To compare from a fairground, we evaluate these pre-trained models with uni-modal input (e_p), and multi-modal input (e_p`<s>` $\mathcal{G}$`<s>` C), separately.

TABLE II: Top-1 accuracies of different models *w.r.t.* their training type, model sizes, input modality.

Training Type	Model Name	# of params (M)	Multi-Modal	Accuracy (%)	
				$B2F_s$	$B2F_m$
From Scratch	LSTM	82.89	✓	6.14	1.04
	Transformer-base	139.22	✓	11.18	6.61
	Transformer-large	406.03	✓	13.40	8.63
	CODIT	105.43	✗	6.53	4.79
Fine-tuned	CodeBERT	172.50	✗	24.28	16.76
			✓	26.05	17.13
	GraphCodeBERT	172.50	✗	24.44	16.85
			✓	25.67	18.31
	CodeGPT	124.44	✗	28.13	16.35
			✓	28.43	17.64
	MODIT	139.22	✗	26.67	19.79
			✓	**29.99**	**23.02**

Results. Table II shows the accuracy in top 1 predicted patch by MODIT along with different baselines. LSTM based $\mathcal{S}2\mathcal{S}$ model predicted 6.14% and 1.04% correct patches in $B2F_s$ and $B2F_m$ respectively. The *Transformer-base* model achieves 11.18% and 6.61% top-1 accuracy in those datasets, which improves further to 13.40% and 8.63% with the *Transformer-large* model. CODIT predicts 6.53% and 4.79% correct patches in $B2F_s$ and $B2F_m$, respectively. Note that CODIT takes the external information in the form of CFG; thus, the patches CODIT generate are syntactically correct. Nevertheless, the transformers, even the smaller model, perform better to predict the correct patch. We conjecture that the transformer model can implicitly learn the code syntax without direct supervision.

In contrast to the models trained from scratch, when we fine-tune a pretrained model, it generates *significantly more* correct patches than models trained from scratch. For instance, MODIT (initialized with pretrained PLBART) generates *168%* and *248%* more correct patches than the *Transformer-base* model (with randomly initialized parameters), despite both of these models having the same architecture and the same number of parameters. In fact, the smallest fine-tuned model (CodeGPT) performs much better than the larger model trained from scratch (*Transformer-large*).

All the fine-tuned models exhibit better performance when the input data are multi-modal with various degrees of improvement. With all three input modalities, CodeBERT [15] generates 7% and 2.2% more correct patches in $B2F_s$ and $B2F_m$, respectively, compared to a unimodal CodeBERT model. In case of MODIT, such improvement is 11.07% in $B2F_s$ and 16.23% in $B2F_m$. The $\mathcal{G}$ in the multi-modal data often contains explicit hints about how to change the code. For instance, consider the example shown in Figure 2, the guidance explicitly says there is a problem with the `json` when it is `empty`. Furthermore, with the presence of C in the input, the model can identify different variables, methods used in the method and potentially copy something from the context. We conjecture that such additional information from these two additional input modalities (i) reduce the search space for change patterns, (ii) help models copy relevant identifiers from the context.

Among the fine-tuned models multi-modalities, MODIT generates 15.12% more correct patches than CodeBERT, 16.82% than GraphCodeBERT, and 5.49% than CodeGPT in $B2F_s$. In the case of $B2F_m$ dataset, MODIT's improvement in performance is 34.38%, 25.72%, 30.50% higher than CodeBERT, GraphCodeBERT, and CodeGPT, respectively. To understand these results better, let us look at some of the examples.

Figure 5 shows an example patch where MODIT correctly generated the expected patch but CodeGPT could not. If we look closely, we can see that the code to be changed (e_p) is a boolean expression where the two clauses are combines with `&&`. While only the first clause, `one.isSimilar(two)` is the expected output, CodeGPT chooses the second clause, `one.toString().equals(two.toString())` from the original. Recall from Figure 4b, CodeGPT processes the com-

```
//Guidance: merging of items that aren't actually equal
public static boolean equals(
                ItemStack one, ItemStack two) {
-   return one.isSimilar(two) &&
-           (one.toString().equals(two.toString()));
+   return one.isSimilar(two); //MODIT generated
    /* CodeGPT generated */
+   return one.toString().equals(two.toString());
}
```

Fig. 5: Example patch where MODIT was able to generate correct patch, but CodeGPT could not. MODIT's patch is shown in green, and CodeGPT generated patch is shown in blue.

bined input and output sequence (separated by special <SEP> token) in *left-to-right* fashion. Thus, encodes representation of the input tokens do not contain information about the whole input sequence. In contrast, the MODIT uses a pre-trained *bi-direction* encoder which helps MODIT to understand the input fully. Based on the examples we have seen and the empirical result, we conjecture that, for code-editing tasks, the model must fully understand the input in a bi-directional fashion.

```
// Guidance: ... code refactoring ...
public boolean isEmpty() {
-  if((first) == null){ return true;}
-  return false;
+  return (first) == null; //MODIT predicts
   /* CodeBERT generated */
+  return ((first) == null) || (first.get()) == null;
}
```

Fig. 6: Correctly predicted patch by MODIT. CodeBERT could not understand and reason about the textual hint to predict the correct patch.

Figure 6 shows an example where MODIT generated correct patch, CodeBERT could not. Note that the guidance text explicitly asks about *code refactoring*, implying that the patched code should be semantically similar to the original code. Similar to the original code, patched could should return `true` when `first == null`, otherwise it should return `false`. An automated code change tool should not add additional code features when doing the refactoring. However, CodeBERT generated patch which introduced an additional clause `first.get()== null` in the return expression, which make CodeBERT's generate code semantically different from the original. MODIT was able to generate the correct patch for this example.

Finally, we summarize the empirical lessons we learned in this research question as

- Multi-modal input improves Code-Editing capability, irrespective of the underlying model used. The guidance often narrows the edit pattern search space, and the context narrows down the token generation search space.
- Transformer models (especially larger ones) are robust enough to learn the code's syntax information without direct supervision. When a pre-trained model is used to initialize transformer parameters, the improvement is *notably higher*.
- For code-editing task, both *understanding the input* and *correctly generated* output are important. While a pre-trained encoder understands the code and a pre-trained decoder generates correct code, an end-to-end pre-trained encoder-decoder model (*e.g.,* PLBART) the best choice to fine-tune for this task.

Result 1: MODIT *generates 29.99%, and 23.02% correct patches in top-1 position for two different datasets outperforming CodeBERT by up to 25.72%, GraphCodeBERT by up to 34.38%, and CodeGPT by up to 30.50%. Pre-trained models tend to be more effective than models trained from scratch for code editing—*MODIT *improves the performance by 167% than the best model trained from the scratch.*

MODIT combines multiple modalities of information to generate patches. Now we investigate,

RQ2. What are the contribution of different input modalities in MODIT 's performance?

Experimental Setup. In this experiment, we investigate the contribution of different input modalities in MODIT's performance. Recall from Section III-A that we use three inputs in MODIT (*i.e.,* e_p, C, $\mathcal{G}$). Here, we investigate different combinations of such input modalities. More precisely, we investigate the influence of three information sources: (i) code that needs to be changed (e_p), (ii) context (C), and (iii) guidance ($\mathcal{G}$). Note that, by presenting e_p as a separate information modality, we are essentially providing MODIT with the information about the location of the change. To study the effect of such presentation, we study another alternative experimental setup, where we annotate the change location inside the context with two unique tokens <START> and <END>.

TABLE III: Contribution of different input modalities in MODIT's performance. ✓ indicates that corresponding input modality is used as encoder input, ✗ indicates otherwise. We report top-1 accuracy as performance measure. Exp. ID is used later to refer to corresponding experiment result. Exp. ID Φ_* denotes an experiment with $*$ as input modalities.

Exp. ID	Inputs			Accuracy (%)	
	e_p	C	$\mathcal{G}$	$B2F_s$	$B2F_m$
Φ_c	✗	✓	✗	13.05	4.50
Φ_{cg}	✗	✓	✓	**17.89**	**4.51**
$\Phi_c^\dagger$	✗	✓†	✗	13.03	4.53
$\Phi_{cg}^\dagger$	✗	✓†	✓	**17.90**	**4.60**
Φ_e	✓	✗	✗	26.67	19.79
Φ_{eg}	✓	✗	✓	**28.76**	**21.63**
Φ_{ec}	✓	✓	✗	29.79	21.40
Φ_{ecg}	✓	✓	✓	**29.99**	**23.02**

† e_p is surrounded by two special tokens **<START>** and **<END>** inside the context.

Result. Table III shows MODIT's performance with different combination of input modalities. When we present only the context to MODIT, it predicts 13.05% correct patches in $B2F_s$ and 4.50% in the $B2F_m$, which improves further to 17.89%, and 4.51% in those two datasets respectively when we add $\mathcal{G}$. Note that in these two scenarios, the model does not explicitly know which portion of the code needs to be edited; it sees the

whole method and predicts (only) the patched code (e_n). In addition to learning how to patch, the model implicitly learns where to apply the patch in this setup. To test whether the identification of such location is the performance bottleneck, we surround the code that needs to be patched with two special tokens <START> and <END>. SequenceR [11] also proposed such annotation of buggy code. Surprisingly, such annotation resulted in comparable (slightly worse in one case) performance by MODIT.

In the next set of experiments, we extract the code that needs to be edited (e_p) and present it as a separate input modality. First, we only present the e_p without the other two modalities. When we only present the e_p and generate the edited code (e_n), it results in 26.67% top-1 accuracy in the $B2F_s$ and 19.79% in the $B2F_m$. Ding *et al.* [32] attributed such improvement to the reduced search space due to shorter input. Our result corroborates their empirical findings. Nevertheless, when we add the $\mathcal{G}$ modality with the e_p, MODIT's performance improves to 28.76% and 21.63% in $B2F_s$ and $B2F_m$, respectively.

In our final set of experiments in this research question, we augment e_p with the C. In this evaluation setup, MODIT predicts 29.79% correct patches in the $B2F_s$ and 21.40% in the $B2F_m$, which is improved further to 29.99%, and 23.02% correct patches in those two datasets when we add $\mathcal{G}$.

```
// Guidance: fixed some bugs in type checking
// improved performance by caching types of expressions
private TypeCheckInfo getType(SadlUnionType expression){
   ...
   return new TypeCheckInfo(
-        declarationConceptName, declarationConceptName
         /* MODIT generated patch with guidance */
+        declarationConceptName, declarationConceptName,
+        this, expression
         /* MODIT generated patch without guidance */
+        this.declarationConceptName,
+        this.declarationConceptName
   );
}
```

Fig. 7: Example showing the effect of textual guidance in MODIT's performance. MODIT produced the correct patch with guidance, without guidance as input MODIT's produced patch is essentially refactored version of original input.

Figure 7 shows an example where MODIT with all modalities could successfully generate correct patch. The text guidance ($\mathcal{G}$) provides hint that variable `expression` should somehow associate with the construction of `TypeCheckInfo` in the **patched code**. However, without this guidance MODIT generated a **wrong patch** by accessing existing parameters from **`this`** object. Essentially, without the guidance, MODIT refactored the input code.

Figure 8 shows the effect of context as input modality to MODIT. The before edit version of the code(e_p) passed the wrong parameter (`m`) to `sendMessage` function. When the context (C) is presented to MODIT, it saw another variable (`sent`) in the context. In contrast, without context(C), MODIT indeed changed the parameter; but sent `m.toString()` — resulting in a wrong patch.

```
// Guidance: Fix bug of sending wrong message
public void setPredecessor (model.Message m) {
    this.predecessor = Integer.valueOf(m.Content);
    model.Message sent = new model.Message();
    sent.To = m.Origin;
-   sendMessage(m);
    /* MODIT generates with the context. */
+   sendMessage(sent);
    /* MODIT generates without context as input. */
+   sendMessage(m.toString());
}
```

Fig. 8: Example showing the necessity of context information in predicting the correct patch. MODIT's generated correct patch with the context as input. Without context, MODIT received `sendMessage(m)` and the guidance as input, did not know the variable `sent` could be the parameter of the function `sendMessage`, and predicted a wrong patch.

When we extract the buggy code and present the buggy code along with the context, we see a big performance improvement (see the difference between Φ_c, and Φ_{ec} in Table III). We hypothesize that, when only context (*i.e.,* full code) is presented (Φ_c), the model gets confused to identify which portion from the context needs to be edited since any portion of the code is a likely candidate for patching. However, when we extract the exact code that needs to be edited and present as a separate input modality to MODIT, it can focus on patching just that code using other modalities (including the context) as a supporting source of information. In a recent study, Ding *et al.* [32] pointed out the need for effective ways to include context in the NMT based code editors. Our empirical results show that MODIT's way of including context as a separate modality is a potential solution to that problem.

In summary, each of the modalities contribute to the overall performances of MODIT. Lessons learned in these experiments are:

- Additional textual guidance helps the patch generation. Such guidance can provide important clue about how to modify the code and sometimes provide ingredients necessary for the change.
- Adding context explicitly in the input enables the model to select appropriate identifiers for patching.
- Isolating buggy code help the model put proper focus on the necessary part of the code while leveraging auxiliary information from other modalities.

Result 2: *All three modalities (code to be edited, context, and guidance) are essential for* MODIT *to perform the best. Without either one of those, performance decreases.* MODIT*'s performance improves up to 37.37% when additional textual guidance is used as an input modality. Context modality improves* MODIT*'s performance up to 6.4%.*

We investigate alternative ways to combine multiple input modalities. We ask,

RQ3. What is the best strategy to encode multiple input modalities?

Experimental setup. To validate MODIT's design choice of appending all input modalities into one sequence, we test

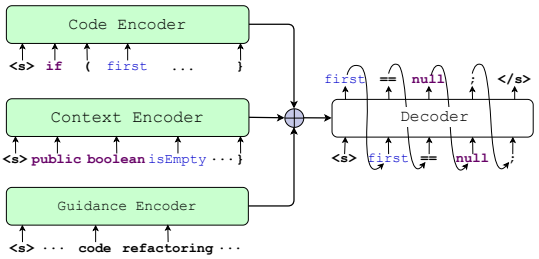


Fig. 9: An alternative architecture of code editing with multi-encoder model. We initialize each of the encoders with pre-trained Encoder model.

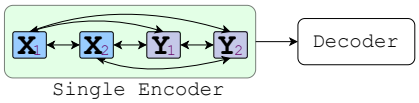
alternative ways to combine input modalities. In particular, we follow the design choice proposed by Lutellier *et al.* [10], where they used multiple encoders to encode the e_p and the C. Tufano *et al.* [33] also leverages a similar idea to encode input code and code review messages. Nevertheless, we use a multi-encoder model shown in Figure 9. In a multi-encoder setting, we first encode each input modality with a corresponding dedicated encoder. After the encoder finishes encoding, we concatenate the encoded representations and pass those to the decoder for generating patched code. To retain maximum effectiveness, we initialize each individual encoder with pretrained weights from CodeBERT [15]. We consider a single-encoder model (also initialized with CodeBERT) as a baseline to compare on the fairground. While presenting the inputs to the single encoder model, we concatenate input modalities with a unique separator token `<s>`. Finally, to test the robustness of our empirical finding, we propose two different experimental settings. In the first evaluation setup, we use all three input modalities. We compare a tri-encoder model with a single-encoder model. Next, we consider bimodal input data – e_p and $\mathcal{G}$. We use a dual-encoder model and compare it with a single-encoder model in this setup.

TABLE IV: Comparison of multi encoder model.

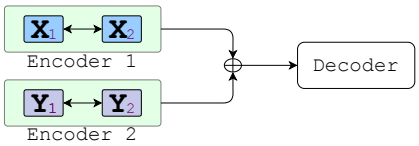
# of Modalities	# of Encoders	Accuracy (%)	
		$B2F_s$	$B2F_m$
3 (e_p, $\mathcal{G}$, C)	3	20.63	11.69
	1	**26.05**	**17.13**
2 (e_p, $\mathcal{G}$)	2	23.12	15.49
	1	**23.81**	**17.46**

Result. Table IV shows the result of multi-encoder models. For tri-modal input data, if we use three different encoders, the model can predict 20.63% correct patches in the $B2F_s$ and 11.69% correct patches in the $B2F_m$. In contrast, if we use a single encoder, the model's predictive performance increases to 26.05% and 17.13% top-1 accuracy in the $B2F_s$ and the $B2F_m$, respectively.

In the bimodal dataset (where the input modalities are e_p and $\mathcal{G}$), the dual-encoder model predicts 23.12% correct patches in the top-1 position for the $B2F_s$ and 15.49% correct for the $B2F_m$. The single encoder counterpart, in this setup, predicts 23.81% correct patches for the $B2F_s$ and 17.46% for the $B2F_m$. The empirical results show that the single-encoder model performs better in both the experimental setup than the multi-encoder setup. We find similar results with GraphCodeBERT [16].



(a) Single encoder for encoding multiple-modalities. Encoder can learn representation *w.r.t.* all modalities.



(b) Dual-encoder for encoding individual modalities separately. Representation of tokens from a particular modality is learned *w.r.t.* (only) other tokens from the same modality.

Fig. 10: Input token representation generation in single encoder and multiple encoder.

To explain why single-encoder is performing better than multi-encoder, let us look at the encoders' working procedure. Figure 10 depicts how the encoder generates representation for input tokens. Note that the encoders we used in this research question are transformer-based, and recall from the Section II, transformer generates representation for an input token by learning its dependency on all other tokens in the sequence. When we present all the input modalities to a single encoder, it generates input representation for those tokens *w.r.t.* and other tokens in the same modality and tokens from other modalities. For instance, in Figure 10a, the encoder generates X_2's representation considering X_1, Y_1, and Y_2. In contrast, in Figure 10, X_2's representation is learned only *w.r.t.* X_1, since encoder1 does not see the input modality Y. Thus, when we present all the input modalities to one single encoder, we conjecture that learned representations are more robust than that of learning with multi-encoder.

Finally, we summarize the lessons we learned in this research question as

- In multi-modal translation, using single encoder results in better performance than using a separate encoder for each modality.
- Single-encoder generates input representation by inter-modality reasoning (attention), hence learns more robust representation than that of multi-encoder.

Result 3: *Encoding all the input modalities by a single encoder is the best way to learn in a multi-modal setting. A single encoder improves code-editing performance by up to 46.5% than the corresponding multi-encoder setting.*

VI. Discussion

A. Localization of Code Edit Site

An alternative modeling approach for code editing is to generate the sequence of edit operations (*i.e.,* `INSERT`, `DELETE`, `UPDATE`) [32], [34]–[36], where the model must know the precise location of an edit operation (often a node in

the AST) before applying it. Throughout this paper, we also assumed that such edit location is known to MODIT. This assumption may pose a threat to the usefulness of MODIT in a real development scenario. To mitigate such a threat, we perform an experiment where we pass the whole function as input to MODIT and expect the whole edited function to be generated. Table V shows the top-1 accuracy in the $B2F_s$ and the $B2F_m$. MODIT generates correctly patched full code in 20.35% cases for the $B2F_s$ and 8.35% cases for the $B2F_m$. With additional textual guidance, the performance is further improved to 21.57% and 13.18% in the $B2F_s$ and $B2F_m$, respectively. While textual guidance helps in this experimental setup, we notice a big drop in performance than the results shown in Table III. This is because the benchmark datasets we used contain small edits (see Table I). Thus, while generating the full code, the model wastes a large amount of effort trying to generate things that did not change. Nevertheless, our hypothesis *external guidance improves code editing* holds even when the model generates full code.

TABLE V: Performance of MODIT when the input in the full code and the output is patched full code.

Inputs		Accuracy (%)	
Full Code	Guidance	$B2F_s$	$B2F_m$
✓	✗	20.35	8.35
✓	✓	**21.57**	**13.18**

B. Tokenization for Source Code Processing

TABLE VI: Comparison between concrete tokenization and abstract tokenization alongside pre-trained models. Results are shown as top-1 accuracy of full code generation in $B2F_s$/ $B2F_m$ datasets.

Token type	CodeBERT	GraphCodeBERT	PLBART
Abstract	16.4 / 5.16	**17.30** / **9.10**	19.21 / **8.98**
Concrete	**17.3** / **8.38**	16.65 / 8.64	**20.35** / 8.35

The possible number of source code can be virtually infinite. Vocabulary explosion has been a big challenge while processing source code with Machine Learning technique [7], [37]. Previous research efforts have addressed this problem using several different heuristics. For instance, Tufano *et al.* [7], [8] identifiers abstraction, which drastically reduces the vocabulary size considered making it easier to learn patterns by the model. Recent studies [9], [10], [32], [37] found that Byte-Pair Encoding [38] partially solves the open-vocabulary problem by sub-dividing rare words into relatively less rare sub-words. Such sub-division is also learned from large corpora of data. All the pre-trained models used in this paper used sub-word tokenization techniques. CodeBERT and GraphCodeBERT used RoBERTa tokenizer [39], CodeGPT used GPT tokenizer [40], and PLBART used sentence-piece tokenizer [28]. The use of such tokenizers strips away the burden of identifier abstraction. Our investigation shows that, in some cases, pre-trained models perform better with concrete tokens than abstract tokens (see Table VI for detailed result). Thus, we champion using input and outputs with concrete tokens when a pre-trained model is used.

VII. RELATED WORKS

A. Automatic Code Change

There are a lot of research efforts to capture repetitiveness of developers' way of editing source code. These researches show the potential of automatic refactoring [41], [42], boilerplate code [43] etc. These research efforts include (semi-)automatic tools involving traditional program analysis techniques (*e.g.,* clone detection, dependency analysis, graph matching) [3], [44]. Other research direction aims at learning source code edit from previous edits and applying those edit patterns in similar context [1], [45]. Some of these efforts targets very specific code changes; For example, Nguyen *et al.* [46] proposed a graph-matching-based approach for automatically updating API usage. Tansey *et al.* [47] semantic preserving transformation of java classes for automated refactoring. Other directions of works address more general-purpose code change learned from open source repositories [5], [6]. Such approaches target solving automated code editing tasks in a data-driven approach, and the edit patterns are learned from example changes. In this research, we also investigated general purpose source code changes in the wild. More closely to MODIT, Rolim *et al.* [4]'s proposed technique constraints source code generation with additional input/output specification or test cases. Nevertheless, we argue that textual guidance could be a very good surrogate specification.

B. NMT for Code Change Modeling

NMT has been studied for past couple of years to learn automatic source code change modeling. Tufano *et al.* [5], [7], [8] presented initial investigation of using NMT in learning general purpose code changes. Chakraborty *et al.* [6] proposed a tree based hierarchical NMT model for general purpose source code change. Instead of viewing code as sequence of tokens, they first generated syntax tree by sampling from Context Free Grammar, and then another model to fill up the gaps for identifier. To reduce the search space, they performed scope analysis to search for suitable identifier. Chen *et al.* [11] proposed a copy mechanism based NMT model for APR where the input is the code before change along with the context, and the output is the code after change. Their work treated the input as uni-modal way where the whole code is one singe modality. In this work, we consider multi-modal way of modeling, where we isolate the code fragment that needs to be changed from its context and present that code fragment concatenated with context to the model. Lutellier *et al.* [10] treated code needs to be changed and the context as two difference modalities and use separate encoders. However, our empirical evidence showed that using one encoder to encode all the modalities result in the best performance. More recently, Ding *et al.* [32] presented empirical evidence that instead of generating a whole code element (*i.e.,* context+change) of the target version, only generating the sequence of changes might perform better for code change modeling. Recent works [35],

[36] proposed models for generating such edit sequence. Such models may augment or outperform NMT based code editing – we leave such investigation as future work.

C. Machine Learning for Source Code Analysis

In recent years, Machine Learning, especially Deep Learning has been widely adopted across different area of software engineering due to Availability of large collection of source code in open source platforms (*e.g.,* GitHub, Bitbucket, *etc.*) Application of ML based source code analysis include bug detection in code [48], clone detection [49], code completion [50], vulnerability detection [51], code summarization [21], code translation [52], etc. Recent works also approached to learn general purpose transferable representation learning for source code, which can later be used for various source code related tasks [9], [15], [53]. The approaches for learning such transferable representations can be broadly categorized in two ways. The first category of approaches (*e.g.,* Code2Vec [53]) aims at learning explicit representation for tokens in the code. Another category of approaches (*e.g.,* CodeBERT [15]) transfers syntactic and semantic interaction between code components in the form of pre-trained models. In this approach, a model for a specific task is initialized with a general-purpose pre-trained model, trained to understand and generate code. In this paper, we empirically found that such pre-trained models (PLBART) increase accuracy upto 248% in patch generation.

VIII. Threats to Validity

A. External Validity

Bias in the dataset. Both $B2F_s$, and $B2F_m$ are collection of bug-fix commits, and thus there is a threat that these dataset may exhibit specific bias towards bug-fix patches. While the commits in these datasets are filtered and classified as bug fix commits, these changes are made by real developers as part of development life cycle. Unlike other bugfix datasets [54], $B2F_s$ and $B2F_m$ do not isolate the bug. Thus, we conjecture that possibility of existence of any such bias is minimal.

Noise in commit message. We used commit message as a guidance for code editing. While previous research efforts [55], [56] showed that commit messages are very useful to summarize the changes in a commit, other research efforts [57], [58] also elucidated noises present in the commit message. To mitigate this threat, we carefully chose the dataset we tested MODIT on. The original authors [8] of the the dataset reported that they carefully investigated the dataset and after manual investigation, they reported that 97.6% of the commits in their datasets are true positive. Despite this threat, MODIT's performance seems to improve with commit message as additional input.

B. Construct Validity

In general, developers write commit message after they edited the code, in theory, summarizing the edits they made. In this paper, we assumed an experimental setup where developer would write the summary before editing the code. Such assumption may pose a threat to the applicability of MODIT in real world, since in some cases, the developer may not know what edits they are going to make prior to the actual editing. Regardless, we consider MODIT as a proof-of-concept, where empirically we show that, if a developer had the idea of change in mind, that could help an automated code editor.

C. Internal Validity

All Deep Learning based techniques are sensitive to hyper-parameters. Thus using a sub-optimal hyper-parameter can pose a threat to the validity of MODIT, especially while comparing with other baselines. As we compared with other pre-trained models, we cannot really modify the architecture and dimensions of other pre-trained models. As for other hyper-parameters (*i.e.,* learning rate, batch size, *etc.*), we use the exact same hyper-parameters described by respective paper. Nevertheless, we open source out code and data for broader dissemination.

IX. Conclusion

In this paper, we highlight that an automatic code edit tool should possess knowledge about the underlying programming language, in general. Also, it can benefit from additional information such as edit context and developers' intention expressed in natural language. To that end, we design, present, and evaluate MODIT– a multi-modal NMT-based automated code editor. Our in-depth evaluation shows that MODIT improves code-editing by leveraging knowledge about programming language through pre-training. In addition, we showed that leveraging additional modalities of information could benefit the source code editor. Our empirical evaluation reveals some critical lessons about the design choices of building an automated code editor that we believe will guide future research in automatic code editing.

Acknowledgement

This work is supported in part by NSF grants SHF-2107405, SHF-1845893, IIS-2040961, IBM, and VMWare. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect those of the US Government, NSF, IBM or VMWare.

References

[1] B. Ray, M. Nagappan, C. Bird, N. Nagappan, and T. Zimmermann, "The uniqueness of changes: Characteristics and applications," ser. MSR '15. ACM, 2015.

[2] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: generating program transformations from an example," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 329–342, 2011.

[3] ——, "Lase: Locating and applying systematic edits by learning from examples," *In Proceedings of 35th International Conference on Software Engineering (ICSE)*, pp. 502–511, 2013.

[4] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 404–415.

[5] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," *arXiv preprint arXiv:1901.09102*, 2019.

[6] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "Codit: Code editing with tree-based neural models," *IEEE Transactions on Software Engineering*, vol. 1, pp. 1–1, 2020.

[7] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," 2018.

[8] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[9] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," *arXiv preprint arXiv:2103.00073*, 2021.

[10] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 101–114.

[11] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, 2019.

[12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30*, 2017, pp. 5998–6008.

[13] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: How far are we?" in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 373–384.

[14] W. Wang, G. Li, S. Shen, X. Xia, and Z. Jin, "Modular tree network for source code representation learning," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–23, 2020.

[15] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, Nov. 2020, pp. 1536–1547.

[16] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, J. Yin, D. Jiang *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *International Conference on Learning Representations*, 2021.

[17] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *2021 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2021.

[18] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *International Conference on Learning Representations*, 2015.

[19] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, 2017, pp. 440–450.

[20] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," in *Advances in Neural Information Processing Systems 32*, 2019, pp. 6563–6573.

[21] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2020.

[22] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," *arXiv preprint arXiv:2003.13848*, 2020.

[23] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[24] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Pre-trained contextual embedding of source code," *arXiv preprint arXiv:2001.00059*, 2019.

[25] L. Dong, N. Yang, W. Wang, F. Wei, X. Liu, Y. Wang, J. Gao, M. Zhou, and H.-W. Hon, "Unified language model pre-training for natural language understanding and generation," *arXiv preprint arXiv:1905.03197*, 2019.

[26] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021. [Online]. Available: https://arxiv.org/abs/2102.04664

[27] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *arXiv preprint arXiv:1910.13461*, 2019.

[28] T. Kudo and J. Richardson, "SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Nov. 2018, pp. 66–71.

[29] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code!= big vocabulary: Open-vocabulary models for source code," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1073–1085.

[30] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 313–324.

[31] R. Müller, S. Kornblith, and G. Hinton, "When does label smoothing help?" *arXiv preprint arXiv:1906.02629*, 2019.

[32] Y. Ding, B. Ray, P. Devanbu, and V. J. Hellendoorn, "Patching as translation: the data and the metaphor," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 275–286.

[33] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards automating code review activities," *arXiv preprint arXiv:2101.02518*, 2021.

[34] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations*, 2019.

[35] D. Tarlow, S. Moitra, A. Rice, Z. Chen, P.-A. Manzagol, C. Sutton, and E. Aftandilian, "Learning to fix build errors with graph2diff neural networks," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 19–20.

[36] Z. Yao, F. F. Xu, P. Yin, H. Sun, and G. Neubig, "Learning structural edits via incremental tree transformations," *arXiv preprint arXiv:2101.12087*, 2021.

[37] R. M. Karampatsis and C. Sutton, "How often do single-statement bugs occur? The ManySStuBs4J dataset," *arXiv preprint arXiv:1905.13334*, 2019.

[38] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.

[39] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019. [Online]. Available: https://arxiv.org/abs/1907.11692

[40] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[41] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 211–221.

[42] V. Raychev, M. Schäfer, M. Sridharan, and M. Vechev, "Refactoring with synthesis," in *ACM SIGPLAN Notices*, vol. 48, no. 10. ACM, 2013, pp. 339–354.

[43] N. Meng, L. Hua, M. Kim, and K. S. McKinley, "Does automated refactoring obviate systematic editing?" in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 392–402.

[44] R. Robbes and M. Lanza, "Example-based program transformation," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2008, pp. 174–188.

[45] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2013, pp. 180–190.

[46] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to API usage adaptation," in *ACM Sigplan Notices*, vol. 45, no. 10. ACM, 2010, pp. 302–321.

[47] W. Tansey and E. Tilevich, "Annotation refactoring: inferring upgrade transformations for legacy applications," in *ACM Sigplan Notices*, vol. 43, no. 10. ACM, 2008, pp. 295–312.

[48] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the" naturalness" of buggy code," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 428–439.

[49] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 261–271.

[50] M. R. Parvez, S. Chakraborty, B. Ray, and K.-W. Chang, "Building language models for text with named entities," 2018.

[51] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[52] H. Xu, S. Fan, Y. Wang, Z. Huang, H. Xu, and P. Xie, "Tree2tree structural language modeling for compiler fuzzing," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2020, pp. 563–578.

[53] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," in *Proceedings of the ACM on Programming Languages*, vol. 3. ACM, 2019, p. 40. [Online]. Available: https://doi.org/10.1145/3290353

[54] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 437–440.

[55] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems*, vol. 32, 2019, pp. 10 197–10 207.

[56] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 914–919.

[57] K. Gallaba, C. Macho, M. Pinzger, and S. McIntosh, "Noise and heterogeneity in historical build data: an empirical study of travis ci," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 87–97.

[58] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 481–490.