CODIT: Code Editing with Tree-Based Neural Models

Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray

Abstract—The way developers edit day-to-day code tends to be repetitive, often using existing code elements. Many researchers have tried to automate repetitive code changes by learning from specific change templates which are applied to limited scope. The advancement of deep neural networks and the availability of vast open-source evolutionary data opens up the possibility of automatically learning those templates from the wild. However, deep neural network based modeling for code changes and code in general introduces some specific problems that needs specific attention from research community. For instance, compared to natural language, source code vocabulary can be significantly larger. Further, good changes in code do not break its syntactic structure. Thus, deploying state-of-the-art neural network models without adapting the methods to the source code domain yields sub-optimal results. To this end, we propose a novel tree-based neural network system to model source code changes and learn code change patterns from the wild. Specifically, we propose a tree-based neural machine translation model to learn the probability distribution of changes in code. We realize our model with a change suggestion engine, CODIT, and train the model with more than 24k real-world changes and evaluate it on 5k patches. Our evaluation shows the effectiveness of CODIT in learning and suggesting patches. CODIT can also learn specific bug fix pattern from bug fixing patches and can fix 25 bugs out of 80 bugs in Defects4J.

✦

1 INTRODUCTION

Developers edit source code to add new features, fix bugs, or maintain existing functionality (*e.g.,* API updates, refactoring, *etc.*) all the time. Recent research has shown that these edits are often repetitive [1], [2], [3]. Moreover, the code components (*e.g.,* token, sub-trees, *etc.*) used to build the edits are often taken from the existing codebase [4], [5]. However, manually applying such repetitive edits can be tedious and error-prone [6]. Thus, it is important to automate code changes, as much as possible, to reduce the developers' burden.

There is a significant amount of industrial and academic work on automating code changes. For example, modern IDEs support specific types of automatic changes (*e.g.,* refactoring, adding boiler-plate code [7], [8], *etc*). Many research tools aim to automate some types of edits, *e.g.,* API related changes [9], [10], [11], [12], [13], [14], refactoring [15], [16], [17], [18], frequently undergone code changes related to Pull Requests [19], *etc*. Researchers have also proposed to automate generic changes by learning either from example edits [20], [21] or from similar patches applied previously to source code [2], [3], [19], [22]. Automated *Code Change*[1] task is defined as modification of existing code (*i.e.,* adding, deleting, or replacing code elements) through applying such frequent change patterns [19], [23], [24], [25]

While the above lines of work are promising and have shown initial success, they either rely on predefined change templates or require domain-specific knowledge: both are hard to generalize to the larger context. However, all of them leverage, in someway, common edit patterns. Given that a large amount of code and its change history is available thanks to software forges like GitHub, Bitbucket, *etc.*, a natural question arises: *Can we learn to predict general code changes by learning them in the wild?*

Recently there has been a lot of interest in using Machine Learning (ML) techniques to model and predict source code from real world [26]. However, modeling changes is different from modeling generic code generation, since modeling changes is conditioned on the previous version of the code. In this work, we investigate whether ML models can capture the repetitiveness and statistical characteristics of code edits that developers apply daily. Such models should be able to automate code edits including feature changes, bug fixes, and other software evolution-related tasks.

A *code edit* can be represented as a tuple of two code versions:$<prev,target>$. To model the edits, one needs to learn the conditional probability distribution of the *target* code version given its *prev* code version. A good probabilistic model will assign higher probabilities to plausible target versions and lower probabilities to less plausible ones. Encoder-decoder Neural Machine Translation models (NMT) are a promising approach to realize such code edit models, where the previous code version is encoded into a latent vector representation. Then, the target version is synthesized (decoded) from the encoded representation. Indeed some previous works [19], [27], [28] use Seq2Seq NMT models to capture changes in token sequences. However, code edits also contain structural changes, which need a syntax-aware model.

To this end, we design a two step encoder-decoder model that models the probability distribution of changes. In the first step, it learns to suggest structural changes in code using a tree-to-tree model, suggesting structural changes in the form of Abstract Syntax Tree (AST) modifications. Tree-based models, unlike their token-based counterparts, can capture the rich structure of code and always produce syntactically correct patches. In the second step, the model concretizes the previously generated code fragment by predicting the tokens conditioned on the AST that was generated in the first step: given the type of each leaf node in the syntax tree, our model suggests concrete tokens of the correct type while respecting scope information. We combine these two models to realize CODIT, a code change suggestion engine, which accepts

---

• *Saikat Chakraborty, Yangruibo Ding, and Baishakhi Ray are affiliated with Department of Computer Science, Columbia University, New York, NY 10027.*
E-mail: {saikatc@cs., yangruibo.ding@, rayb@cs.}columbia.edu
• *Miltiadis Allamanis is with Microsoft Research, Cambridge, UK.*
E-mail: miallama@microsoft.com.

1. We use the term *Code Change* and *Code Edit* interchangeably

a code fragment and generates potential edits of that snippet.

In this work, we particularly focus on smaller changes as our previous experience [3] shows that such changes mostly go through similar edits. In fact, all the previous NMT-based code transformation works [19], [27], [28] also aim to automate such changes. A recent study by Karampatsis *et al.* [29] showed that small changes are frequent—our analysis of the top 92 projects of their dataset found that, on average, in each project, one line changes take place around 26.71% of the total commits and account for up to 70% for the bug fix changes. Our focus in this work is primarily to automatically change small code fragments (often bounded by small AST sizes and/or few lines long) to reflect such repetitive patterns. Note that, in theory, our approach can be applied to any small fragment of code in the project repository with any programming language. However, for prototyping, we designed CODIT to learn changes that belong to methods of popular java project in Github.

In this work, we collect a new dataset — *Code-Change-Data*, consisting of 32,473 patches from 48 open-source GitHub projects collected from Travis Torrent [30]. Our experiments show CODIT achieves 15.94% patch suggestion accuracy in the top 5 suggestions; this result outperforms a Copy-Seq2Seq baseline model by 63.34% and a Tree2Seq based model by 44.37%.

We also evaluate CODIT on *Pull-Request-Data* proposed by Tufano *et al.* [19]. Our evaluation shows that CODIT suggests 28.87% of correct patches in the top 5 outperforming Copy-Seq2Seq-based model by 9.26% and Tree2Seq based model by 22.92%. Further evaluation on CODIT's ability to suggest bug-fixing patches in Defects4J shows that CODIT suggests 15 complete fixes and 10 partial fixes out of 80 bugs in Defects4J. In summary, our key contributions are:

- We propose a novel tree-based code editing machine learning model that leverages the rich syntactic structure of code and generates syntactically correct patches. To our knowledge, we are the first to model code changes with tree-based machine translation.
- We collect a large dataset of 32k real code changes. Processed version of the dataset is available at https://drive.google.com/file/d/1wSl_SN17tbATqlhNMO0O7sEkH9gqJ9Vr.
- We implement our approach, CODIT, and evaluate the viability of using CODIT for changes in a code, and patching bug fixes. Our code is available at https://git.io/JJGwU.

2 BACKGROUND

Modeling Code Changes. Generating source code using machine learning models has been explored in the past [31], [32], [33], [34]. These methods model a probability distribution $p(c|\kappa)$ where c is the generated code and κ is any contextual information upon which the generated code is conditioned. In this work, we generate *code edits*. Thus, we are interested in models that predict code given its previous version. We achieve this using NMT-style models, which are a special case of $p(c|\kappa)$, where c is the new and κ is the previous version of the code. NMT allows us to represent code edits with a single end-to-end model, taking into consideration the original version of a code and defining a conditional probability distribution of the target version. Similar ideas have been explored in NLP for paraphrasing [35].

Grammar-based modeling. Context Free Grammars (CFG) have been used to describe the syntax of programming languages [36] and natural language [37], [38]. A CFG is a tuple $G = (N, \Sigma, P, S)$ where N is a set of non-terminals, Σ is a set of

terminals, P is a set of production rules in the form of $\alpha \rightarrow \beta$ and $a \in N, b \in (N \cup \Sigma)^*$, and S is the start symbol. A sentence (*i.e.* sequence of tokens) that belongs to the language defined by G can be parsed by applying the appropriate derivation rules from the start symbol S. A common technique for generation of utterances is to expand the left-most, bottom-most non-terminal until all non-terminals have been expanded. Probabilistic context-free grammar (PCFG) is an extension of CFG, where each production rule in associated with a probability, *i.e.* is defined as (N, Σ, P, Π, S) where Π defines a probability distribution for each production rule in P conditioned on α.

Neural Machine Translation Models. NMT models are usually a cascade of an *encoder* and a *decoder*. The most common model is sequence-to-sequence (seq2seq) [39], where the input is treated as a sequence of tokens and is encoded by a sequential encoder (*e.g.,* biLSTM). The output of the encoder is stored in an intermediate representation. Next, the decoder using the encoder output and another sequence model, *e.g.,* an LSTM, generates the output token sequence. In NMT, several improvements have been made over the base seq2seq model, such as *attention* [39] and *copying* [40]. Attention mechanisms allow decoders to automatically search information within the input sequence when predicting each target element. Copying, a form of an attention mechanism, allows a model to directly copy elements of the input into the target. We employ both attention and copying mechanisms in this work.
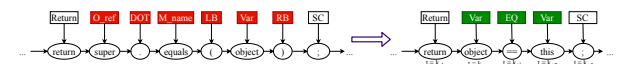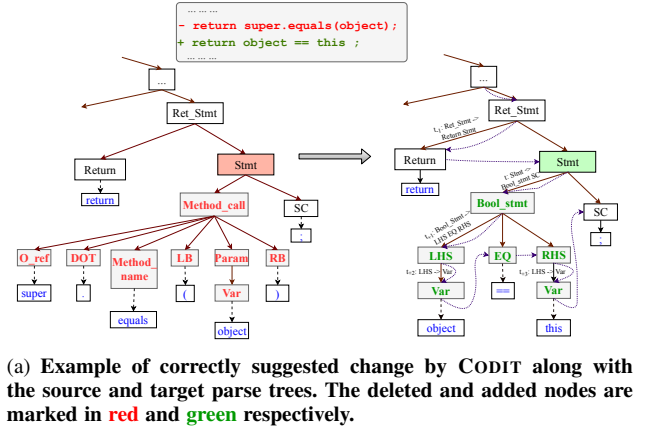
3 MOTIVATING EXAMPLE

(a) **Example of correctly suggested change by** CODIT **along with the source and target parse trees. The deleted and added nodes are marked in red and green respectively.**

(b) **Sequence of grammar rules extracted from the parse trees.**

(c) **Token generation. Token probabilities are conditioned based on terminal types generated by tree translator (see figure 1(a))**

Fig. 1: **Motivating Example**

Figure 1 illustrates an example of our approach. Here, the original code fragment `return super.equals(object)` is edited to `return object == this`. CODIT takes these two code fragments along with their context, for training. While suggesting changes, *i.e.*, during test time, CODIT takes as input the previous version of the code and generates its edited version.

CODIT operates on the parse trees of the previous (t_p) and new (t_n) versions of the code, as shown in Figure 1(a) (In the rest of the paper, a subscript or superscript with p and n correspond to previous and new versions respectively). In Figure 1, changes are applied only to the subtree rooted at the $Method\_call$ node. The subtree is replaced by a new subtree (t_n) with Bool_stmt as a root. The deleted and added subtrees are highlighted in **red** and **green** respectively.

While modeling the edit, CODIT first predicts the structural changes in the parse tree. For example, in Figure 1(a) CODIT first generates the changes corresponding to the subtrees with dark nodes and red edges. Next the structure is concretized by generating the token names (terminal nodes). This is realized by combining two models: (i) a tree-based model predicting the structural change (see 4.1) followed by a (ii) a token generation model conditioned on the structure generated by the tree translation model (see 4.2).

Tree Translator. The tree translator is responsible for generating structural changes to the tree structure. A machine learning model is used to learn a (probabilistic) mapping between t_p and t_n. First, a tree encoder, encodes t_p computing a distributed vector representation for each of the production rules in t_p yielding the distributed representation for the whole tree. Then, the tree decoder uses the encoded representations of t_p to sequentially select rules from the language grammar to generate t_n. The tree generation starts with the root node. Then, at each subsequent step, the bottom-most, left-most non-terminal node of the current tree is expanded. For instance, in Figure 1(a), at time step t, node Stmt is expanded with rule **Stmt → Bool_Stmt** SC. When the tree generation process encounters a terminal node, it records the node type to be used by the token generation model and proceeds to the next non-terminal. In this way, given the LHS rule sequences of Figure 1(b) the RHS rule sequences is generated.

Token Generator: The token generator predicts concrete tokens for the terminal node types generated in the previous step. The token generator is a standard seq2seq model with attention and copying [39] but constrained on the token types generated by the tree translator. To achieve this, the token generator first encodes the token string representation and the node type sequence from t_p. The token decoder at each step probabilistically selects a token from the vocabulary or copies one from the input tokens in t_p. However, in contrast to traditional seq2seq where the generation of each token is only conditioned on the previously generated and source tokens, we additionally condition on the token type that has been predicted by the tree model and generate only tokens that are valid for that toke type. Figure 1(c) shows this step: given the token sequence of the original code < super . equals (object) > and their corresponding token types (given in dark box), the new token sequence that is generated is < object == this >.

4 TREE-BASED NEURAL TRANSLATION MODEL

We decompose the task of predicting code changes in two stages: First, we learn and predict the structure (syntax tree) of the edited code. Then, given the predicted tree structure, we concretize the code. We factor the generation process as

$$P(c_n|c_p) = P(c_n|t_n, c_p)P(t_n|t_p)P(t_p|c_p) \quad (1)$$

and our goal is to find $\hat{c}_n$ such that $\hat{c}_n = argmax_{c_n} P(c_n|c_p)$. Here, c_p is the previous version of the code and t_p is its parse tree, whereas c_n is the new version of the code and t_n its parse

tree. Note that parsing a code fragment is unambiguous, *i.e.* $P(t_p|c_p) = 1$. Thus, our problem takes the form

$$\hat{c}_n = \arg\max_{c_n, t_n} \underbrace{P(c_n|t_n, c_p)}_{\mathscr{M}_{token}} \cdot \underbrace{P(t_n|t_p)}_{\mathscr{M}_{tree}} \quad (2)$$

Equation 2 has two parts. First, it estimates the changed syntax tree $P(t_n|t_p)$. We implement this with a tree-based encoder-decoder model (section 4.1). Next, given the predicted syntax tree t_n, we estimate the probability of the concrete edited code with $p(c_n|t_n, c_p)$ (Section 4.2).

4.1 Tree Translation Model ($\mathscr{M}_{tree}$)

The goal of $\mathscr{M}_{tree}$ is to model the probability distribution of a new tree (t_n) given a previous version of the tree (t_p). For any meaningful code the generated tree is syntactically correct. We represent the tree as a sequence of grammar rule generations following the CFG of the underlying programming language. The tree is generated by iteratively applying CFG expansions at the left-most bottom-most non-terminal node (frontier_node) starting from the start symbol.

For example, consider the tree fragments in Figure 1(a). Figure 1(b) shows the sequence of rules that generate those trees. For example, in the right tree of Figure 1(a), the node Ret_Stmt is first expanded by the rule: Ret_Stmt→Return Stmt. Since, Return is a terminal node, it is not expanded any further. Next, node Stmt is expanded with rule: Stmt→Bool_Stmt SC. The tree is further expanded with Bool_Stmt→LHS EQ RHS, LHS→Var, and RHS→Var. During the tree generation process, we apply these rules to yield the tree fragment of the next version.

In particular, the tree is generated by picking CFG rules at each non-terminal node. Thus, our model resembles a Probabilistic Context-Free Grammar (PCFG), but the probability of each rule depends on its surroundings. The neural network models the probability distribution, $P(R_k^n|R_1^n, ...R_{k-1}^n, t_p)$: At time k the probability of a rule depends on the input tree t_p and the rules $R_1^n, ...R_{k-1}^n$ that have been applied so far. Thus, the model for generating the syntax tree t_n is given by

$$P(t_n|t_p) = \prod_{k=1}^{\tau} P(R_k^n|R_1^n, ...R_{k-1}^n, t_p) \quad (3)$$

Encoder: The encoder encodes the sequence of rules that construct t_p. For every rule R_i^p in t_p, we first transform it into a single learnable distributed vector representation $\boldsymbol{r}_{R_i^p}$. Then, the LSTM encoder summarizes the whole sequence up to position i into a single vector $\boldsymbol{h}_i^p$.

$$\boldsymbol{h}_i^p = f_{LSTM}(\boldsymbol{h}_{i-1}^p, \boldsymbol{r}_{R_i^p}) \quad (4)$$

This hidden vector contains information about the particular rule being applied and the previously applied rules. Once all the rules in t_p are processed, we get a final hidden representation ($\boldsymbol{h}_\tau^p$). The representations at each time step ($\boldsymbol{h}_1^p, \boldsymbol{h}_2^p, ..., \boldsymbol{h}_\tau^p$) are used in the decoder to generate rule sequence for the next version of the tree. The parameters of the LSTM and the rules representations $\boldsymbol{r}_{R_i^p}$ are randomly initialized and learned jointly with all other model parameters.

Decoder: Our decoder has an LSTM with an attention mechanism as described by Bahdanau *et al.* [39]. The decoder LSTM is initialized with the final output from the encoder, *i.e.* $\boldsymbol{h}_0^n = \boldsymbol{h}_\tau^p$.

At a given decoding step k the decoder LSTM changes its internal state in the following way,

$$\boldsymbol{h}_k^n = f_{LSTM}(\boldsymbol{h}_{k-1}^n, \boldsymbol{\psi}_k), \tag{5}$$

where $\boldsymbol{\psi}_k$ is computed by the attention-based weighted sum of the inputs $\boldsymbol{h}_j^p$ as [39] in , *i.e.*

$$\boldsymbol{\psi}_k = \sum_{j=1}^{\tau} softmax(\boldsymbol{h}_{k-1}^{n}{}^T \boldsymbol{h}_j^p)\boldsymbol{h}_j^p \tag{6}$$

Then, the probability over the rules at the kth step is:

$$P(R_k^n | R_1^n, ... R_{k-1}^n, t_p) = softmax(W_{tree} \cdot \boldsymbol{h}_k^n + \mathbf{b}_{tree}) \tag{7}$$

At each timestep, we pick a derivation rule R_k^n following equation (7) to expand the `frontier_node` (n_f^t) in a depth-first, left-to-right fashion. When a terminal node is reached, it is recorded to be used in $\mathscr{M}_{token}$ and the decoder proceeds to next non-terminal. In Equation (7), W_{tree} and $\mathbf{b}_{tree}$ are parameters that are jointly learned along with the LSTM parameters of the encoder and decoder.

4.2 Token Generation Model ($\mathscr{M}_{token}$)

We now focus on generating a concrete code fragment c, *i.e.* a sequence of tokens $(x_1, x_2, ...)$. For the edit task, the probability of an edited token x_k^n depends not only on the tokens of the previous version $(x_1^p, ..., x_m^p)$ but also on the previously generated tokens $x_1^n, ..., x_{k-1}^n$. The next token x_k^n also depends on the token type (θ), which is generated by $\mathscr{M}_{tree}$. Thus,

$$P(c_n | c_p, t_n) = \prod_{k=1}^{m'} P(x_k^n | x_1^n, ..., x_{k-1}^n, \{x_1^p, ..., x_m^p\}, \theta_k^n) \tag{8}$$

Here, θ_k^n is the node type corresponding to the generated terminal token x_k^n. Note that, the token generation model can be viewed as a conditional probabilistic translation model where token probabilities are conditioned not only on the context but also on the type of the token (θ_*^*). Similar to $\mathscr{M}_{tree}$, we use an encoder-decoder. The encoder encodes each token and corresponding type of the input sequence into a hidden representation with an LSTM (figure 1(c)). Then, for each token (x_i^p) in the previous version of the code, the corresponding hidden representation (s_i^p) is given by: $\boldsymbol{s}_i^p = f_{LSTM}(\boldsymbol{s}_{i-1}^p, enc([x_i^p, \theta_i^p]))$. Here, θ_i^p is the terminal token type corresponding to the generated token x_i^p and $enc()$ is a function that encodes the pair of x_i^p, θ_i^p to a (learnable) vector representation.

The decoder's initial state is the final state of the encoder. Then, it generates a probability distribution over tokens from the vocabulary. The internal state at time step k of the token generation is $\boldsymbol{s}_k^n = f_{LSTM}(\boldsymbol{s}_{k-1}^n, enc(x_k^n, \theta_k^n), \boldsymbol{\xi}_k))$, where $\boldsymbol{\xi}_k$ is the attention vector over the previous version of the code and is computed as in Equation (6). Finally, the probability of the kth target token is computed as

$$\begin{aligned} P(x_k^n &| x_1^n, ..., x_{k-1}^n, \{x_1^p, ..., x_m^p\}, \theta_k^n) \\ &= softmax\left(W_{token} \cdot \boldsymbol{s}_k^n + \boldsymbol{b}_{token} + mask(\theta_k^n)\right) \end{aligned} \tag{9}$$

Here, W_{token} and $\boldsymbol{b}_{token}$ are parameters that are optimized along with all other model parameters. Since not all tokens are valid for all the token types, we apply a mask that deterministically filters out invalid candidates. For example, a token type of

boolean_value, can only be concretized into **true** or **false**. Since the language grammar provides this information, we create a mask ($mask(\theta_k^n)$) that returns a $-\infty$ value for masked entries and zero otherwise. Similarly, not all variable, method names, type names are valid at every position. We refine the mask based on the variables, method names and type names extracted from the scope of the change. In the case of method, type and variable names, CODIT allows $\mathscr{M}_{token}$ to generate a special `<unknown>` token. However, the `<unknown>` token is then replaced by the source token that has the highest attention probability (*i.e.* the highest component of $\boldsymbol{\xi}_k$), a common technique in NLP. The mask restricts the search domain for tokens. However, in case to variable, type, and method name $\mathscr{M}_{token}$ can only generate whatever token available to it in the vocabulary (through masking) and whatever tokens are available in input code (through copying).

5 IMPLEMENTATION

Our tree-based translation model is implemented as an edit recommendation tool, CODIT. CODIT learns source code changes from a dataset of patches. Then, given a code fragment to edit, CODIT predicts potential changes that are likely to take place in a similar context. We implement CODIT extending OpenNMT [41] based on PyTorch. We now discuss CODIT's implementation in details.

Patch Pre-processing. We represent the patches in a parse tree format and extract necessary information (*e.g.,* grammar rules, tokens, and token-types) from them.

Parse Tree Representation. As a first step of the training process, CODIT takes a dataset of patches as input and parses them. CODIT works at method granularity. For a method patch Δm, CODIT takes the two versions of m: m_p and m_n. Using GumTree, a tree-based code differencing tool [42], it identifies the edited AST nodes. The edit operations are represented as insertion, deletion, and update of nodes *w.r.t.* m_p. For example, in Figure 1(a), **red** nodes are identified as deleted nodes and **green** nodes are marked as added nodes. CODIT then selects the minimal subtree of each AST that captures all the edited nodes. If the size of the tree exceeds a maximum size of max_change_size, we do not consider the patch. CODIT also collects the edit context by including the nodes that connect the root of the method to the root of the changed tree. CODIT expands the considered context until the context exceed a tree size threshold (max_tree_size). During this process, CODIT excludes changes in comments and literals. Finally, for each edit pair, CODIT extracts a pair (AST_p, AST_n) where AST_p is the original AST where a change was applied, and AST_n is the AST after the changes. CODIT then converts the ASTs to their parse tree representation such that each token corresponds to a terminal node. Thus, a patch is represented as the pair of parse trees (t_p, t_n).

Information Extraction. CODIT extracts grammar rules, tokens and token types from t_p and t_n. To extract the rule sequence, CODIT traverses the tree in a depth-first pre-order way. From t_p, CODIT records the rule sequence ($R_1^p, ..., R_\tau^p$) and from t_n, CODIT gets ($R_1^n, ..., R_{\tau'}^n$) (Figure 1(b)). CODIT then traverses the parse trees in a pre-order fashion to get the augmented token sequences, *i.e.* tokens along with their terminal node types: (x_*^p, θ_*^p) from t_p and (x_*^n, θ_*^n) from t_n. CODIT traverses the trees in a left-most depth-first fashion. When a terminal node is visited, the corresponding augmented token (x_*^*, θ_*^*) is recorded.

Model Training. We train the tree translation model ($\mathscr{M}_{tree}$) and token generation model ($\mathscr{M}_{token}$) to optimize Equation (3)

TABLE 1: **Summary of datasets used to evaluate CODIT.**

Dataset	# Projects	# Train Examples	# Validtion Examples	# Test Examples	# Tokens		# Nodes	
					Max	Average	Max	Average
Code-Change-Data	48	24072	3258	5143	38	15	47	20
Pull-Request-Data [28]	3	4320	613	613	34	17	47	23
Defects4J-data [44]	6	22060	2537	117	35	16	43	21

and Equation (8) respectively using the cross-entropy loss as the objective function. Note that the losses of the two models are independent and thus we train each model separately. In our preliminary experiment, we found that the quality of the generated code is not entirely correlated to the loss. To mitigate this, we used top-1 accuracy to validate our model. We train the model for a fixed amount of n_{epoch} epochs using early stopping (with patience of $valid_{patience}$) on the top-1 suggestion accuracy on the validation data. We use stochastic gradient descent to optimize the model.

Model Testing. To test the model and generate changes, we use beam-search [43] to produce the suggestions from $\mathcal{M}_{tree}$ and $\mathcal{M}_{token}$. First given a rule sequence from the previous version of the tree, CODIT generates K_{tree} rule sequences. CODIT subsequently use these rule sequence to build the actual AST. While building the tree from the rule sequence, CODIT ignores the sequence if the rule sequence is infeasible (*i.e.*, head of the rule does not match the `frontier_node`, n_f^t). Combining the beam search in rule sequence and the tree building procedure, CODIT generate different trees reflecting different structural changes. Then for each tree, CODIT generates K_{token} different concrete code. Thus, CODIT generates $K_{tree} \cdot K_{token}$ code fragments. We sort them based on their probability, *i.e.* $log(P(c_n|c_p, t_p)) = log(P(c_n|c_p, t_n) \cdot P(t_n|t_p))$. From the sorted list of generated code, we pick the top K suggestions.

6 EXPERIMENTAL DESIGN

We evaluate CODIT for three different types of changes that often appear in practice: (i) code change in the wild, (ii) pull request edits, and (iii) bug repair. For each task, we train and evaluate CODIT on different datasets. Table 1 provides detailed statistics of the datasets we used.

(i) *Code Change Task.* We collected a large scale real code change dataset (*Code-Change-Data*) from 48 open-source projects from GitHub. These projects also appear in TravisTorrent [30] and have aat least 50 commits in Java files. These project are excludes any forked project, toy project, or unpopular projects (all the projects have at least 10 watchers in Github). Moreover, these projects are big and organized enough that they use Travis Continuous integration system for maintainability. For each project, we collected the revision history of the main branch. For each commit, we record the code before and after the commit for all Java files that are affected. In total we found java 241,976 file pairs. We then use GumTree [42] tool to locate the change in the file and check whether the changes are inside a method in the corresponding files. Most of the changes that are outside a method in a java class are changes related to import statements and changes related to constant value. We consider those out of CODIT's scope. We further remove any pairs, where the change is only in literals and constants. Excluding such method pairs, we got 182,952 method pairs. We set $max\_change\_size = 10$ and $max\_tree\_size = 20$. With this data collection hyper-parameter settings, we collected 44,382 patches.

We divide every project based on their chronology. From every project, we divide earliest 70% patches into train set, next 10% into validation set and rest 20% into test set based on the project chronology. We removed any exact test and validation examples from the training set. We also removed intra-set duplicates. After removing such duplicate patches, we ended up with 32,473 patches in total, which are then used to train and evaluate CODIT.

(ii) *Pull Request Task.* For this task, we use *Pull-Request-Data*, provided by Tufano *et al.* [19] which contains source code changes from merged pull requests from three projects from Gerrit [45]. Their dataset contains 21774 method pairs in total. Similar to the *Code-Change-Data*, we set $max\_change\_size = 10$ and $max\_tree\_size = 20$ to extract examples that are in CODIT's scope from this dataset. We extracted 5546 examples patch pair.

(iii) *Bug Repair Task.* For this task, we evaluate CODIT on Defects4J [44] bug-fix patches. We extract 117 patch pairs across 80 bug ids in Defect4j dataset with the same $max\_change\_size = 10$ and $max\_tree\_size = 20$ limit as Code Change Task. These are the bugs that are in CODIT's scope. To train CODIT for this task, we create a dataset of code changes from six projects repositories in Defects4J dataset containing 24597 patches. We remove the test commits from the training dataset.

6.1 Evaluation Metric

To evaluate CODIT, we measure for a given code fragment, how accurately CODIT generates patches. We consider CODIT to correctly generate a patch if it exactly matches the developer produced patches. CODIT produces the top K patches and we compute CODIT's accuracy by counting how many patches are correctly generated in top K. Note that this metric is stricter than semantic equivalence.

For the bug fix task, CODIT takes a buggy line as input and generates the corresponding patched code. We consider a bug to be fixed if we find a patch that passes all the test cases. We also manually investigate the patches to check the similarity with the developer provided patches.

6.2 Baseline

We consider several baselines to evaluate CODIT's performance. Our first baseline in a vanilla LSTM based Seq2Seq model with attention mechanism [39]. Results of this baseline indicate different drawbacks of considering raw code as a sequence of token.

The second baseline, we consider, is proposed by Tufano *et al.* [27]. For a given code snippet (previous version), Tufano *et al.* first abstracted the identifier names and stored the mapping between original and abstract identifiers in a symbol table. The resultant abstracted code (obtained by substituting the raw identifiers with abstract names) is then translated by an NMT model. After translation, they concretize the abstract code using the information stored in the symbol table. The patches where the abstract symbols predicted by the model are not found in the symbol table remian undecidable. Such patches, although can

be useful to guide developers similar to our $\mathcal{M}_{tree}$, cannot be automatically concretized, and thus, we do not count them as fully correct patches.

Both the vanilla Seq2Seq and Tufano *et al.*'s model consider the before version of the code as input. Recently, SequenceR [28] proposed way to represent additional context to help the model generate concrete code. We design such a baseline, where we add additional context to c_P. Following SequenceR, we add copy attention, where the model learns to copy from the contexed code.

To understand the tree encoding mechanism, we used several tree encoders. First method we considered is similar to DeepCom [46], where the AST is represented as a sequential representation called Structure Based Traversal (SBT). Second tree encoding method we consider is similar to Code2Seq, where code AST is represented by a set of paths in the tree. While these tree encoding methods are used for generating Code comment, we leverage these encoding methods for code change prediction. We design a Seq2Seq method with DeepCom encoder (Tree2Seq), and Code2Seq encoder. We also enable the copy attention in both of these baselines.

We also did some experiment on Kommrusch *et al.* [47]'s proposed Graph2Seq network, which they provided implementation as an add on in OpenNMT-Py framework[2]. However, their implementation is assumed upon a very restricted vocabulary size. To understand better, we modified their code to fit into our context, but neither their original model, nor our modified model works in program change task. When we increase the vocabulary size in their version of the model, it does not scale and quickly exhaust all the memory in the machine (we tried their model on a machine with 4 Nvidia TitanX GPU and 256 GB RAM). When we tested our modification of their code, the model does not converge even after 3 days of training, eventually resulting in 0 correct predictions.

The basic premise of CODIT is based on the fact that code changes are repetitive. Thus, another obvious baseline is to see how CODIT performs *w.r.t.* to code-clone based edit recommendation tool [3]. In particular, given a previous version (c_p) of an edit, we search for the closest k code fragments using similar bag-of-words at the token level similar to Sajnani *et al.* [48]. In our training data of code edits, this step searches in the previous code versions and use the corresponding code fragments of the next version as suggested changes.

Bug-fixing baselines: For the bug fix task, we compare CODIT's performance with two different baselines. Our first baseline is SequenceR [28], we compare with the results they reported. We also compare our result with other non-NMT based program repair systems.

7 RESULTS

We evaluate CODIT's performances to generate concrete patches *w.r.t.* generic edits (RQ1) and bug fixes (RQ3). In RQ2, we present an ablation study to evaluate our design choices.

RQ1. How accurately can CODIT suggest concrete edits?

To answer this RQ, we evaluate CODIT's accuracy *w.r.t.* the evaluation dataset containing concrete patches. Table 2 shows the results: for *Code-Change-Data*, CODIT can successfully generate 201 (3.91%), 571 (11.10%), and 820 (15.94%) patches at top 1, 2, and 5 respectively. In contrast, at top 1, SequenceR generates 282 (5.48%) correct patches, and performs the best among all the

2. https://opennmt.net/OpenNMT-py/ggnn.html

methods. While SequenceR outperforms CODIT in top 1, CODIT outperforms SequenceR with significant margin at top 2 and top 5.

In *Pull-Request-Data*, CODIT generates 57 (9.3%), 134 (21.86%), and 177 (28.87%) correct patches at top 1, 2, and 5 respectively. At top 1, Tufano *et al.*'s [27] method produces 81 patches. At top 2, CODIT produces 134 (21.86%) patches, which is comparable with SequenceR's result 137 (22.35%). At top 5, CODIT outperforms all the other baselines achieving 9.2% gain over SequenceR baseline.

The main advantage point of CODIT is that, since it considers the structural changes separate from the token changes, it can learn the structural change pattern well instead of being dominated by learning the code token changes. However, being a two stage process, CODIT has two different hinge point for failure. If $\mathcal{M}_{tree}$ does not generate the correct tree, no matter how good $\mathcal{M}_{token}$ performs, CODIT is unable to generate correct patch. We conjecture that, this is the reason for CODIT's failure at top 1.

Among the baselines we compared here, SequenceR, and Tree2Seq takes the advantage of copy attention. Tufano *et al.*'s model takes the advantage of reduced vocabulary through identifier abstraction. Tufano *et al.*'s model works best when the code is mostly self contained *i.e.*, when there is always a mapping from abstract identifier to concrete identifier in the symbol table, their method can take full advantage of the vocabulary reduction. to When the input code is mostly self contained (*i.e.*, most of the tokens necessary to generate a patch are inside the input code c_p or appears within the context limit in as presented to SequenceR).

In code change task with NMT, the deterministic positions of code tokens are important to put attention over different parts of code. While Code2Seq [49]'s representation of input code as random collection of paths in the code showed success for code comprehension, it does not generalize well for code change due to the stochastic nature of the input. Additionally, copy attention cannot be trivially applied to Code2Seq since, like attention, copy also rely on the defined positions of tokens in the input.

While CODIT replaces any `<unknown>` prediction by the token with highest attention score (see section 4.2), unlike SequenceR, CODIT does not learn to copy. The rationale is, unlike SequenceR, CODIT's input code (c_p) is not enhanced by the context. Instead, we present the context to the CODIT through the token mask (see equation (9)). If we enable copy attention, CODIT becomes highly biased by the tokens that are inside c_p.
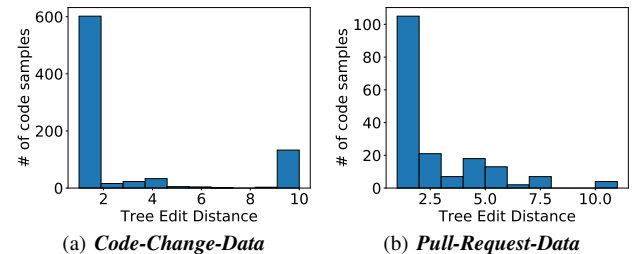

(a) *Code-Change-Data* (b) *Pull-Request-Data*

Fig. 2: **Patch size (Tree Edit-distance) histogram of correctly generated patches in different datasets.**

Note that, *"vocabulary explosion"* still remains an open problem for code generation. Neither CODIT nor any other baselines we discussed here solve this problem. CODIT presents a way to learn the structural change and restricts the search domain for token names through a mask. For instance, where $\mathcal{M}_{token}$ needs

TABLE 2: **Performance of CODIT suggesting concrete patches. For Token Based models, predominant source of information in the code are token sequences. For Tree Based models information source is code AST. For IR based method, information retrieval model is used on code.**

Method		Code Change Data			Pull Request Data		
		Number of examples : 5143			Number of examples : 613		
		Top-1	Top-2	Top-5	Top-1	Top-2	Top-5
Token Based	Seq2Seq	107 (2.08%)	149 (2.9%)	194 (3.77%)	45 (7.34%)	55 (8.97%)	69 (11.26%)
	Tufano *et al.*	175 (3.40%)	238 (4.63%))	338 (6.57%)	**81 (13.21%)**	104 (16.97%)	145 (23.65%)
	SequenceR	**282 (5.48%)**	398 (7.74%)	502 (9.76%)	39 (6.36%)	**137 (22.35%)**	162 (26.43%)
Tree Based	Tree2Seq	147 (2.86%)	355 (6.9%)	568 (11.04%)	39 (6.36%)	89 (14.52%)	144 (23.49%)
	Code2seq	58 (1.12%)	82 (1.59%)	117 (2.27%)	4 (0.65%)	7 (1.14%)	10 (1.63%)
	CODIT	201 (3.91%)	**571 (11.10%)**	**820 (15.94%)**	57 (9.3%)	134 (21.86%)	**177 (28.87%)**
IR based	$\mathscr{B}_{ir}$	40 (0.77%)	49 (0.95%)	61 (1.18%)	8 (1.30%)	8 (1.30%)	9 (1.46%)

to generate a token of *primitive data type* ($\mathscr{M}_{token}$ knows the token type because $\mathscr{M}_{tree}$ already generated a tree with terminal node types), it can restrict the search over the primitive types only. While it is expected that the decoder of an ideal Seq2Seq model would inherently learn appropriate tokens at appropriate positions as it implicitly learns code structure, in reality, because of its unrestricted search space, they tend to mispredict more tokens than CODIT.

In general, CODIT along with all the baselines perform better when generating small patches. For example, a large majority of the correctly generated patches have size of one (*i.e.* $\Delta_t = 1$, the tree distance between t_p and t_n [50]). However, a non-trivial number of larger patches are also correctly generated. Figure 2 shows the histogram of the size of correctly predicted patches. For example, there are 202 and 51 correct patches with $\Delta_t \geq 3$ generated by CODIT for *Code-Change-Data* and *Pull-Request-Data* respectively.

For qualitative evaluation, we show some non-trivial patches that CODIT can successfully generate. CODIT can learn a wide range of patch patterns. Table 3 shows few examples of different category of patches that CODIT can generate. CODIT also shows promise in generating non-trivial structure based changes. Consider Example 4 where x is renamed to session both the formal parameter and the usage in the body. Note that, since CODIT uses a tree-based model it is good at capturing long-distance dependencies allowing the token-level model to focus on predicting tokens, *e.g.,* such that it can rename the same variable similarly. Another interesting example where CODIT can successfully generate patches is shown in example 6, where CODIT does not only add the **abstract** keyword in the method signature, but also removes the body. Since CODIT is aware of code syntax, it learns that method declarations with an **abstract** keyword have a high probability of an empty method body.

Table 4 shows some additional examples where CODIT can successfully generate patches. In these examples, different exception/error types (*i.e.* Exception, Error, RuntimeException) are changed to EOFException although their usage differs. In the first three examples EOFException is used as a class reference, while for the others EOFException is used to initialize an object. These examples also illustrate CODIT's ability to generalize to different contexts and use-cases. Other structural transformation that CODIT include, but not limited to, include scoping (example 7 in table 3), adding/deleting method parameters (example 3 in table 3), changing method/variable access modifiers (example 9, 10 in table 3), etc.

Result 1: CODIT *suggests 15.94% correct patches for Code-Change-Data and 28.87% for Pull-Request-Data within top 5 and outperforms best baseline by 44.37% and 9.26% respectively.*

Next, we evaluate CODIT's sub-components.

RQ2. How do different design choices affect CODIT's performance? This RQ is essentially an ablation study where we investigate in three parts: (i) the token generation model ($\mathscr{M}_{token}$), (ii) the tree translation model ($\mathscr{M}_{tree}$), and (iii) evaluate the combined model, *i.e.* CODIT, under different design choices. We further show the ablation study on the data collection hyper-parameters (*i.e.,* max_change_size, and max_tree_size) and investigate the cross project generalization.

Evaluating token generation model. Here we compare $\mathscr{M}_{token}$ with the baseline SequenceR model. Note that $\mathscr{M}_{token}$ generates a token given its structure. Thus, for evaluating the standalone token generation model in CODIT's framework, we assume that the true structure is given (emulating a scenario where a developer knows the kind of structural change they want to apply). Table 5 presents the results.

While the baseline Seq2Seq with copy-attention (SequenceR) generates 9.76% (502 out of 5143) and 26.43% (162 out of 613) correct patches for *Code-Change-Data* and *Pull-Request-Data* respectively at top 5, Table 5 shows that if the change structure (*i.e.* t_n) is known, the standalone $\mathscr{M}_{token}$ model of CODIT can generate 39.57% (2035 out of 6832) and 61.66% (378 out of 613) for *Code-Change-Data* and *Pull-Request-Data* respectively. This result empirically shows that if the tree structure is known, NMT-based code change prediction significantly improves. In fact, this observation led us build CODIT as a two-stage model.

Evaluating tree translation model. Here we evaluate how accurately $\mathscr{M}_{tree}$ predicts the structure of a change — shown in Table 6. $\mathscr{M}_{tree}$ can predict 56.78% and 55.79% of the structural changes in *Code-Change-Data* and *Pull-Request-Data* respectively. Note that, the outputs that are generated by $\mathscr{M}_{tree}$ are not concrete code, rather structural abstractions. In contrast, Tufano *et al.*'s [27] predicts 38.56% and 39.31% correct patches in *Code-Change-Data* and *Pull-Request-Data* respectively. Note that, their abstraction and our abstraction method is completely different. In their case, for some of the identifiers (those already exist in the symbol table), they have a deterministic way to concretize the code. In our case, $\mathscr{M}_{token}$ in CODIT augments $\mathscr{M}_{tree}$ by providing a stochastic way to concretize every identifier by using NMT.

Note that, not all patches contain structural changes (*e.g.,* when

TABLE 3: **Examples of different types of concrete patches generated by CODIT**

Example 1. API Change

```
return f.createJsonParser createParser(...)
```

Example 2. Type Change

```
void appendTo(StringBuffer StringBuilder buffer)
```

Example 3. Parameter: Add/Delete Method Parameter

```
1. testDataPath(false , true , true , true, false );
2. assertNotificationEnqueued(map ,key ,value ,hash)
```

Example 4. Refactoring: Modify Method Parameters Name

```
void visit(JSession x session , ...) throws Exception
{
    visit (((JNode) (x session)), ...);
}
```

Example 5. Statement: Add Statement

```
{...
    interruptenator.shutdown();
    Thread.interrupted();
}
```

Example 6. Inheritance: Abstracting a Method

```
public abstract void removeSessionCookies (...)
{
    throw new android...MustOverrideException();
}
```

Example 7. Exception Change: Add Try Block

```
public void copyFrom( java.lang.Object arr){
+    try{
    android.os.Trace.traceBegin (...);
+    finally{
    android.os.Trace.traceEnd(...);
+    }
}
```

Example 8. Other: Delete Unreferenced Variable

```
public void testConstructor2NPE(){
...
AtomicIntegerArray aa = new AtomicIntegerArray(a);
shouldThrow () ;
...
}
```

Example 9. Final: Make Method Parameter Immutable

```
public String print(final ReadableInstant inst){...}
```

Example 10. Access: Modify Access Type

```
public protected AbstractInstant () { super(); }
```

Every cell shows an example of correctly suggested patches by CODIT. Top line is the patch category, followed by the actual patch. In the patch, **Red** tokens/lines are deleted and **Green** tokens/lines are added.

TABLE 4: **Examples of CODIT's ability to generalize in different use cases.** `Exception`, `Error`, `RuntimeException` **are modified to** `EOFException` **under different context.**

```
Ex:1. return Error EOFException.class ;
Ex:2. return Exception EOFException.class ;
Ex:3. return RuntimeException EOFException.class ;
Ex:4. return new Error EOFException(msg) ;
Ex:5. return new Exception EOFException(msg);
Ex:6. return new RuntimeException EOFException(msg);
```

TABLE 5: **Correct patches generated by the standalone token generation model when the true tree structure is known.**

Dataset	Total Correct Patches	
	SequenceR	standalone $\mathcal{M}_{token}$
Code-Change-Data	502 (9.76%)	2035 (39.57%)
Pull-Request-Data	162 (26.43%)	378 (61.66%)

TABLE 6: $\mathcal{M}_{tree}$ **top-5 performance for different settings.**

Dataset	# Correct Abstract Patches*	
	Tufano *et al.*	CODIT
Code-Change-Data	1983 / 5143 (38.56%)	2920 / 5143 (56.78%)
Pull-Request-Data	241 / 613 (39.31%)	342 / 613 (55.79%)

* Each cell represents correctly predicted patches / total patches (percentage of correct patch) in the corresponding setting.

a separate version of $\mathcal{M}_{tree}$ using only the training patches with at least 1 node differing between t_n and t_p. We also remove examples with no structural changes from the test set. This is our filtered dataset. In the filtered dataset, $\mathcal{M}_{tree}$ predicts 33.61% and 30.73% edited structures from *Code-Change-Data* and *Pull-Request-Data* respectively. This gives us an estimate of how well $\mathcal{M}_{tree}$ can predict *structural* changes.

Evaluating CODIT. Having $\mathcal{M}_{tree}$ and $\mathcal{M}_{token}$ evaluated separately, we will now evaluate our end-to-end combined model ($\mathcal{M}_{tree}$ + $\mathcal{M}_{token}$) focusing on two aspects: (i) effect of attention-based copy mechanism, (ii) effect of beam size.

First, we evaluate contribution of CODIT's attention-based copy mechanism as described in Section 4.2. Table 7 shows the results. Note that, unlike SequenceR, CODIT is not trained for learning to copy. Our copy mechanism is an auxiliary step in the beam search that prohibits occurrence of `<unknown>` token in the generated code.
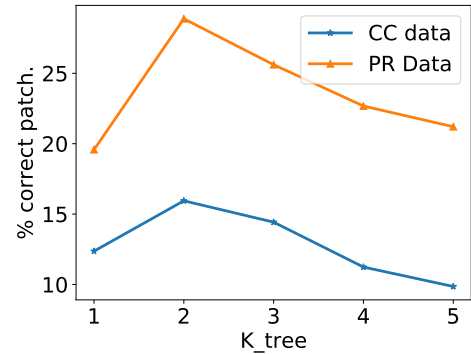
Fig. 3: **Performance of CODIT @top-5 (K_{token} = 5) for different K_{tree}**

Recall that $\mathcal{M}_{token}$ generates a probability distribution over the vocabulary (Section 4.2). Since the vocabulary is generated using the training data, any unseen tokens in the test patches are replaced by a special `<unknown>` token. In our experiment, we found that a large number (about 3% is *Code-Change-Data* and about 4% is *Pull-Request-Data*) of patches contain `<unknown>` tokens; this is undesirable since the generated code will not compile. When we do not replace `<unknown>` tokens, CODIT can predict 742 (14.42%), and 163 (26.59%) correct patches in *Code-Change-Data* and *Pull-Request-Data* respectively. However, if all the `<unknown>` tokens could be replaced perfectly with

a single token, such as a method name, is changed). For example, 3050 test patches of *Code-Change-Data*, and 225 test patches of *Pull-Request-Data* do not have structural changes. When we use these patches to train $\mathcal{M}_{tree}$, we essentially train the model to sometimes copy the input to the output and rewarding the loss function for predicting no transformation. Thus, to report the capability of $\mathcal{M}_{tree}$ to predict structural changes, we also train

TABLE 7: **CODIT performance *w.r.t.* to the attention based copy mechanism @top-5** (K_{tree} =2, K_{token} =5). Lower bound is without copy. The upper bound evaluates with oracle copying predictions for `<unknown>`. For CODIT each `<unknown>` token is replaced by the source token with the highest attention.

Dataset	lower bound	upper bound	CODIT
Code-Change-Data	742 (14.42%)	898 (17.46%)	820 (15.94%)
Pull-Request-Data	163 (26.59%)	191 (31.16%)	177 (28.87%)

the intended token, *i.e.* upper bound of the number of correct patches goes up to 898 (17.46%) and 191 (31.16%) correct patches in *Code-Change-Data* and *Pull-Request-Data* respectively. This shows the need for tackling the `<unknown>` token problem. To solve this, we replace `<unknown>` tokens predicted by $\mathscr{M}_{token}$ with the source token with the highest attention probability following Section 4.2. With this, CODIT generates 820 (15.94%), and 177 (28.87%) correct patches from *Code-Change-Data* and *Pull-Request-Data* respectively (Table 7).

Second, we test two configuration parameters related to the beam size, K_{tree} and K_{token} *i.e.* the number of trees generated by $\mathscr{M}_{tree}$ and number of concrete token sequences generated by $\mathscr{M}_{token}$ per tree (Section 5). While Table 2 shows the effect of different values of K_{token} (*e.g.,*, 1, 2, 5), here we investigate the effect of K_{tree} for a given K_{token}. Figure 3 shows the parameter sensitivity of K_{tree} when K_{token} is set 5. Recall from Section 5, CODIT first generates K_{tree} number of trees, and then generates K_{token} number of code per tree. Among those $K_{tree} * K_{token}$ generated code, CODIT chooses top K_{token} number of code to as final output. In both *Code-Change-Data* (CC data in figure 3), and *Pull-Request-Data* (PR data in figure 3), CODIT performs best when $K_{tree} = 2$. When $K_{tree} = 2$, total generated code is 10, among which CODIT chooses top 5. With increasing number of K_{tree}, CODIT has to choose from increasing number of generated code, eventually hurting the performance of CODIT.

TABLE 8: **Ablation study of** $max\_change\_size(max_c)$**, and** $max\_tree\_size(max_t)$**.**

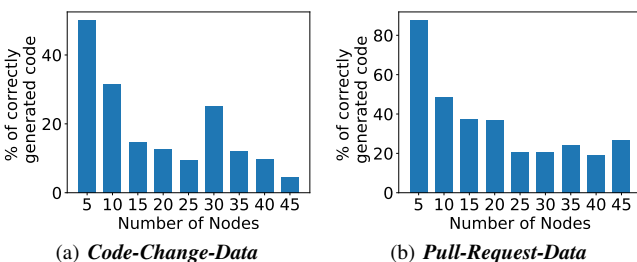| # Train	# Valid	# Test	max_c	max_t	% correct
3988	593	593	10	10	34.06
4320	613	613	10	20	32.63
4559	622	627	10	30	28.07
7069	948	932	20	20	31.55
7340	993	948	20	30	29.32
9361	1227	1213	30	30	24.48

Fig. 4: **Percentage of correct prediction by CODIT with respect to number of nodes in the tree.**

Next, we move onto ablation study of our data collection hyper-parameters (*i.e.*, max_change_size, and max_tree_size). For *Code-Change-Data*, we only collected examples patches where max_change_size = 10, and max_tree_size = 20. For this ablation study, we created 6 different version *Pull-Request-Data* with respective data parameters. As we increase the max_tree_size parameter, the length of the code increases causing the performance to decrease. With the increment of max_change_size, CODIT's performance also decreases. Furthermore, Figure 4 shows histogram of percentage of correctly predicted examples by CODIT *w.r.t.* size of the tree (in terms of nodes). While CODIT performs well in predicting smaller trees ($\leq$ 10 nodes), CODIT also works comparably well in larger tree sizes. In fact, CODIT produces 21.97% correct code in *Pull-Request-Data*, and 16.48% correct code in *Code-Change-Data* where the tree size is larger than 30 nodes.

TABLE 9: **Cross project generalization test for CODIT (% of correct at @top-5).**

Settings	CODIT full	$\mathscr{M}_{tree}$ only
Intra-Project Split	15.94	56.78
Cross Project Split	9.48	59.65

To understand how CODIT generalizes beyond a project, we do a cross-project generalization test. Instead of chronological split of examples (see section 6), we split the examples based on projects, *i.e.*, all the examples that belongs to a project falls into only one split (train/validation/test). We then train and test CODIT based on this data split. Table 9 shows the result in this settings *w.r.t.* to intra-project split. While $\mathscr{M}_{tree}$ in intra-project and cross-project evaluation setting achieves similar performance, full CODIT performance deteriorate by 68%. The main reason behind such deterioration is diverse choice of token name across different projects. Developer tend to use project specific naming convention, api etc. This also indicates that the structural change pattern that developers follow are more ubiquitous across different projects than the token changes.

> **Result 2:** CODIT *yields the best performance with a copy-based attention mechanism and with tree beam size of 2. $\mathscr{M}_{tree}$ achieves 58.78% and 55.79% accuracy and $\mathscr{M}_{token}$ achieves 39.57% and 61.66% accuracy in Code-Change-Data and Pull-Request-Data respectively when tested individually.*

Finally, we evaluate CODIT's ability to fixing bugs.
RQ3. How accurately CODIT suggests bug-fix patches? We evaluate this RQ with the state-of-the-art bug-repair dataset, Defects4J [44] using all six projects.
Training: We collect commits from the projects' original GitHub repositories and preprocess them as described in Section 5. We further remove the Defects4J bug fix patches and use the rest of the patches to train and validate CODIT.
Testing: We extract the methods corresponding to the bug location(s) from the buggy-versions of Defects4J. A bug can have fixes across multiple methods. We consider each method as candidates for testing and extract their ASTs. We then filter out the methods that are not within our accepted tree sizes. In this way, we get 117 buggy method ASTs corresponding to 80 bugs. The rest of the bugs are ignored.

Here we assume that a fault-localization technique already localizes the bug [51]. In general, fault-localization is an integral part of program repair. However, in this paper, we focus on evaluating CODIT's ability to produce patches rather than an end-to-end repair tool. Since fault localization and fixing are methodologically independent, we assume that bug location is given and evaluate whether CODIT can produce the correct patch. Evaluation of CODIT's promise as a full-fledged bug repair tool remains for future work.

For a buggy method, we extract c_p. Then for a given c_p, we run CODIT and generate a ranked list of generated code fragments (c_n). We then try to patch the buggy code with the generated fragments following the rank order, until the bug-triggering test passes. If the test case passes, we mark it a potential patch and recommend it to developers. We set a specific time budget for the patch generation and testing. For qualitative evaluation, we additionally investigate manually the patches that pass the triggering test cases to evaluate the semantic equivalence with the developer-provided patches. Here we set the maximum time budget for each buggy method to 1 hour. We believe this is a reasonable threshold as previous repair tools (*e.g.,* Elixir [52]) set 90 minutes for generating patches. SimFix [53] set 5 hours as their time out for generating patches and running test cases.
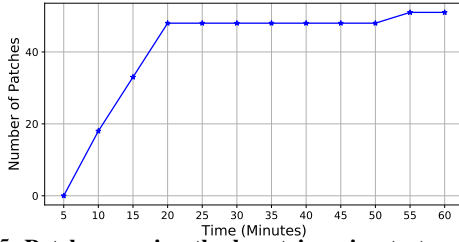

Fig. 5: **Patches passing the bug-triggering tests *v.s.* time.**

CODIT can successfully generate at least 1 patch that passes the bug-triggering test case for 51 methods out of 117 buggy methods from 30 bugs, *i.e.* 43.59% buggy methods are potentially fixed. Figure 5 shows the number of patches passing the bug-triggering test case *w.r.t.* time. We see that, 48 out of 51 successful patches are generated within 20 minutes.

We further manually compare the patches with the developer-provided patches: among 51 potential patches, 30 patches are identical and come from 25 different bug ids (See Table 10). The bugs marked in green are completely fixed by CODIT with all their buggy methods being successfully fixed. For example, Math-49 has 4 buggy methods, CODIT fixes all four. For the bugs marked in blue†, CODIT fixes all the methods that are in scope. For example, for Lang-4, there are 2 methods to be fixed, 1 of them are in CODIT's scope, and CODIT fixes that. However, for two other bugs (marked in orange*), CODIT produces only a partial fix. For example, in the case of Math-46 and Mockito-6, although all the methods are within scope, CODIT could fix 1 out of 2 and 2 out of 20 methods respectively. The 'Patch Type' column further shows the type of change patterns.

SequenceR [28] is a notable NMT based program repair tool which takes the advantage of *learning to copy* in NMT. They evaluated on 75 one line bugs in Defects4J dataset and reported 19 plausible and 14 fully correct successful patches. Among those 75 bugs, 38 are in CODIT's scope. Out of those 38 bugs, CODIT can successfully generate patches for 14 bugs. Note that, we do not present CODIT as full fledged automated program repair

TABLE 10: **CODIT's performance on fixing Defects4J [44] bugs.**

Project	BugId	# methods to be patched	# methods in scope	# methods CODIT can fix	Patch Type
Chart	8	1	1	1	Api Change
	10	1	1	1	Method-Invocation
	11	1	1	1	Variable-Name-Change
	12	1	1	1	Api-Change
Closure	3†	2†	1†	1†	Method-Invocation†
	75†	2†	1†	1†	Return-Value-Change†
	86	1	1	1	Boolean-Value-Change
	92	1	1	1	Api-Change†
	93	1	1	1	Api-Change
Lang	4†	2†	1†	1†	Method-Invocation†
	6	1	1	1	Method-Parameter-Change
	21	1	1	1	Method-Parameter-Change
	26	1	1	1	Method-Parameter-Add
	30†	5†	1†	1†	Type-Change†
Math	6†	13†	1†	1†	Method-Parameter-Change†
	30	1	1	1	Type-Change
	46*	2*	2*	1*	Ternary-Statement-Change*
	49	4	4	4	Object-Reference-Change
	57	1	1	1	Type-Change
	59	1	1	1	Ternary-Statement-Change
	70	1	1	1	Method-Parameter-Add
	98	2	2	2	Array-Size-Change
Mockito	6*	20*	20*	2*	Api-Change*
	25†	6†	1†	1†	Method-Parameter-Add†
	30†	2†	1†	1†	Method-Parameter-Add†

Green rows are bug ids where CODIT can produce complete patch. Blue† rows are where CODIT can fix all the methods that are in CODIT's scope. Orange* rows are where CODIT could not fix all that are in CODIT's scope.

tool, rather a tool for guiding developers. Thus, for automatic evaluation, we assumed the correct values of constants (of any data type) given.

One prominent bug repair approach [52], [54], [55] is to transform a suspicious program element following some change patterns until a patch that passes the test cases is found. For instance, Elixir [52] used 8 predefined code transformation patterns and applied those. In fact, CODIT can generate fixes for 8 bugs out of 26 bugs that are fixed by Elixir [52].

Nevertheless, CODIT can be viewed as a transformation schema which automatically learns these patterns without human guidance. We note that CODIT is *not* explicitly focused on bug-fix changes since it is trained with generic changes. Even then, CODIT achieves good performance in Defects4J bugs. Thus, we believe CODIT has the potential to complement existing program repair tools by customizing the training with previous bug-fix patches and allowing to learn from larger change sizes. Note that, current version of CODIT does not handle multi-hunk bugs. Even if a bug is multi-hunk, in current prototype, we consider each of the hunk as separate input to CODIT. For instance, consider Math-46, which is a 2-hunk bug. While all 2 methods are in CODIT's scope, CODIT can only fix one. Currently we do not consider interaction between multiple hunks [56]. We leave the investigation of NMT in such scenario for future work.

> **Result 3:** CODIT *generates complete bug-fix patches for 15 bugs and partial patches for 10 bugs in Defects4J.*

8 THREATS TO VALIDITY

External Validity. We built and trained CODIT on real-world changes. Like all machine learning models, our hypothesis is that the dataset is representative of real code changes. To mitigate this threat, we collected patch data from different repositories and different types of edits collected from real world.

Most NMT based model (or any other text decoder based models) faces the "vocabulary explosion" problem. That problem is even more prominent in code modeling, since possible names of identifiers can be virtually infinite. While this problem is a major bottleneck in DL base code generation, CODIT does not solve this problem. In fact, similar to previous researches (*i.e.*, SequenceR [28]), CODIT cannnot generate new identifiers if it is not in the vocabulary or in the input code.

Internal Validity. Similar to other ML techniques, CODIT's performance depends on hyperparameters. To minimize this threat, we tune the model with a validation set. To check for any unintended implementation bug, we frequently probed our model during experiments and tested for desirable qualities. In our evaluation, we used exact similarity as an evaluation metric. However, a semantically equivalent code may be syntactically different, *e.g.,* refactored code. We will miss such semantically equivalent patches. Thus, we give a lower bound for CODIT's performance.

9 RELATED WORK

Modeling source code. Applying ML to source code has received increasing attention in recent years [26] across many applications such as code completion [11], [31], bug prediction [57], [58], [59], clone detection [60], code search [61], *etc*. In these work, code was represented in many form, *e.g.,* token sequences [31], [32], parse-trees [62], [63], graphs [59], [64], embedded distributed vector space [65], *etc*. In contrast, we aim to model code changes, a problem fundamentally different from modeling code.

Machine Translation (MT) for source code. MT is used to translate source code from one programming language into another [66], [67], [68], [69]. These works primarily used Seq2Seq model at different code abstractions. In contrast, we propose a syntactic, tree-based model. More closely to our work, Tufano *et al.* [19], [27], and Chen *et al.* [28] showed promising results using a Seq2Seq model with attention and copy mechanism. Our baseline Seq2Seq model is very similar to these models. However, Tufano *et al.* [19], [27] employed a different form of abstraction: using a heuristic, they replace most of the identifiers including variables, methods and types with abstract names and transform previous and new code fragments to abstracted code templates. This vaguely resembles CODIT's $\mathscr{M}_{tree}$ that predicts syntax-based templates. However, the abstraction method is completely different. With their model, we achieved 39% accuracy at top 5. As Table 6, our abstract prediction mechanism also predicts 55-56% accurately on different datasets. Since, our abstraction mechanism differs significantly, directly comparing these numbers does *not* yield a fair comparison. Gupta *et al.* used Seq2Seq models to fix C syntactic errors in student assignments [70]. However, their approach can fix syntactic errors for 50% of the input codes *i.e.* for rest of the 50% generated patches were syntactically incorrect which is never the case for CODIT because of we employ a tree-based approach. Other NMT application in source code and software engineering include program comprehension [46], [49], [71], [72], commit message generation [73], [74], program synthesis [62], [75] etc.

Structure Based Modeling of Code. Code is inherently structured. Many form of structured modeling is used in source code over the years for different tasks. Allamanis *et al.* [76], [77] proposed statistical modeling technique for mining source code idioms, where they leverages probabilistic Tree Substitution Grammar (pTSG) for mining code idioms. CODIT's $\mathscr{M}_{tree}$ is based on similar concept, where we model the derivation rule sequence based on a probabilistic Context Free Grammar. Brockschmidt *et al.* [78], Allmanis *et al.* [59] proposed graph neural network for modeling source code. However, their application scenario is different from CODIT's application, *i.e.* their focus is mainly on generating natural looking code and/or identify bugs in code. Recent researches that are very close to CODIT include Yin *et al.* [79]'s proposed graph neural network-based distributed representation for code edits but their work focused on change representation than generation. Other recent works that focus on program change or program repair include Graph2Diff by Tarlow *et al.* [80], Hoppity by Dinella *et al.* [81]. These research results are promising and may augment or surpass CODIT's performance, but problem formulation between these approach are fundamentally different. While these technique model the change only in the code, we formulate the problem of code change in encoder-decoder fashion, where encoder-decoder implicitly models the changes in code.

Program Repair. Automatic program repair is a well-researched field, and previous researchers proposed many generic techniques for general software bugs repair [55], [82], [83], [84], [85]. There are two differnt directions in program repair research : generate and validate approach, and sysnthesis bases approach. In generate and validate approaches, candidate patches are first generated and then validated by running test cases [52], [55], [86], [87], [88]. Synthesis based program repair tools synthesizes program elements through symbolic execution of test cases [89], [90]. CODIT can be considered a program generation tool in generate and validate based program-repair direction. Arcuri *et al.* [91], Le Goues *et al.* [86] built their tool for program repair based on this assumption. Both of these works used existing code as the search space of program fixes. Elixir [52] used 8 predefined code transformation patterns and applied those to generate patches. CapGen [88] prioritize operator in expression and fix ingredients based in the context of the fix. They also relied on predefined transformation patterns for program mutation. In contrast, CODIT learns the transformation patterns automatically. Le *et al.* [92] utilized the development history as an effective guide in program fixing. They mined patterns from existing change history and used existing mutation tool to mutate programs. They showed that the mutants that match the mined patterns are likely to be relevant patch. They used this philosophy to guide their search for program fix. The key difference between Le *et al.* and this work, is that we do not just mine change patterns, but learn a probabilistic model that learns to generalize from the limited data.

Automatic Code Changes. Modern IDEs [7], [8] provide support for automatic editings, *e.g.,* refactoring, boilerplate templates (*e.g.,* try-catch block) *etc*. There are many research on automatic and semi-automatic [93], [94] code changes as well: *e.g.,* given that similar edits are often applied to similar code contexts, Meng *et al.* [20], [22] propose to generate repetitive edits using code clones, sub-graph isomorphisms, and dependency analysis. Other approaches mine past changes from software repositories and suggest edits that were applied previously to similar contexts [2], [3]. In contrast, CODIT generates edits by learning them from the

wild—it neither requires similar edit examples nor edit contexts. Romil *et al.* [21] propose a program synthesis-based approach to generate edits where the original and modified code fragments are the input and outputs to the synthesizer. Such patch synthesizers can be thought of as a special kind of model that takes into account additional specifications such as input-output examples or formal constraints. In contrast, CODIT is a statistical model that predicts a piece of code given only historical changes and does not require additional input from the developer. Finally, there are domain-specific approaches, such as error handling code generation [95], [96], API-related changes [9], [10], [11], [12], [13], [14], [97], automatic refactorings [15], [16], [17], [18], *etc*. Unlike these work, CODIT focuses on general code changes.

10 DISCUSSION AND FUTURE WORK

Search Space for Code Generation. Synthesizing patches (or code in general) is challenging [98]. When we view code generation as a sequence of token generation problem, the space of the possible actions becomes too large. Existing statistical language modeling techniques endorse the action space with a probability distribution, which effectively reduces the action space significantly since it allows to consider only the subset of probable actions. The action space can be further reduced by relaxing the problem of concrete code generation to some form of abstract code generation, *e.g.,* generating code sketches [97], abstracting token names [27], *etc*. For example, Tufano *et al.* reduce the effective size of the action space to $3.53 \cdot 10^{10}$ by considering abstract token names [27]. While considering all possible ASTs allowed by the language's grammar, the space size grows to $1.81 \cdot 10^{35}$. In this work, a probabilistic grammar further reduces the effective action space to $3.18 \cdot 10^{10}$, which is significantly lower than previous methods. Such reduction of the action space allows us to search for code more efficiently.

Ensemble Learning for Program Repair. The overall performance of pre-trained deep-learning models may vary due to the different model architectures and hyper-parameters, even if they are trained on the same training corpus. Moreover, bug fixing patterns are numerous and highly dependent on the bug context and the bug type, so a single pre-trained model may only have the power to fix certain kinds of bugs and miss the others. To overcome this limitation, ensemble learning can be a potential approach to leverage the capacities of different models and learn the fixing patterns in multiple aspects [99]. In future work, we plan to conduct researches on exploring the potentials of ensemble model to improve the performance of CODIT.

Larger Code Edits. Our work has focused on generating small code changes (single-line or single-hunk) since such changes take a non-trivial part of software evolution. However, predicting larger (multi-line and multi-hunk) code changes is important and always regarded as a harder task for current automated program repair techniques. Generating larger code snippets will significantly increase the difficulty of repairing bugs for pure sequence-to-sequence model, since any wrongly predicted token along the code sequence will lead to meaningless patches. CODIT can address this problem as it takes language grammar into consideration. Specifically, the tree translation model could maintain its power when dealing with larger code changes, because the structural changes are much simpler and more predictable than token-level code changes. Given the tree structure of the patch, CODIT will not widely search for tokens in the whole vocabulary, but rather, only possible candidates corresponding to a node type will be considered. Therefore, such hierarchical model may have potential to generate larger edits. We plan to investigate CODIT's ability to multi-line and multi-hunk code edits in the future.

11 CONCLUSION

In this paper, we proposed and evaluated CODIT, a tree-based hierarchical model for suggesting eminent source code changes. CODIT's objective is to suggest changes that are similar to change patterns observed in the wild. We evaluate our work against a large number of real-world patches. The results indicate that tree-based models are a promising tool for generating code patches and they can outperform popular seq2seq alternatives. We also apply our model to program repair tasks, and the experiments show that CODIT is capable of predicting bug fixes as well.

REFERENCES

[1] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 315–324.

[2] 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.

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

[4] M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches," in *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 492–495.

[5] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 306–317.

[6] B. Ray, M. Kim, S. Person, and N. Rungta, "Detecting and characterizing semantic inconsistencies in ported code," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 367–377.

[7] Microsoft, "Visual Studio (https://visualstudio.microsoft.com)." [Online]. Available: https://visualstudio.microsoft.com

[8] E. Foundation, "Eclipse IDE (https://www.eclipse.org)." [Online]. Available: https://www.eclipse.org

[9] 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.

[10] 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.

[11] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Acm Sigplan Notices*, vol. 49, no. 6. ACM, 2014, pp. 419–428.

[12] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S.-C. Khoo, "Semantic patch inference," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*. IEEE, 2012, pp. 382–385.

[13] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in linux device drivers," in *Acm sigops operating systems review*, vol. 42, no. 4. ACM, 2008, pp. 247–260.

[14] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig, "API code recommendation using statistical learning from fine-grained changes," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 511–522.

[15] S. R. Foster, W. G. Griswold, and S. Lerner, "WitchDoctor: IDE support for real-time auto-completion of refactorings," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 222–232.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] N. Meng, M. Kim, and K. S. McKinley, "LASE: Locating and applying systematic edits by learning from examples," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 502–511.

[23] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE transactions on Software Engineering*, vol. 30, no. 9, pp. 574–586, 2004.

[24] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[25] M. Kim, D. Notkin, and D. Grossman, "Automatic inference of structural changes for matching across program versions," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 333–343.

[26] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys*, 2018.

[27] 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.

[28] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *arXiv preprint arXiv:1901.01808*, 2018.

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

[30] M. Beller, G. Gousios, and A. Zaidman, "TravisTorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration," in *Proceedings of the 14th working conference on mining software repositories*, 2017.

[31] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 837–847.

[32] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *SIGSOFT FSE*, 2014, pp. 269–280.

[33] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 763–773.

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

[35] J. Mallinson, R. Sennrich, and M. Lapata, "Paraphrasing revisited with neural machine translation," in *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, vol. 1, 2017, pp. 881–893.

[36] D. E. Knuth, "Semantics of context-free languages," *Mathematical systems theory*, vol. 2, no. 2, pp. 127–145, 1968.

[37] N. Chomsky, "Three models for the description of language," *IRE Transactions on information theory*, vol. 2, no. 3, pp. 113–124, 1956.

[38] J. E. Hopcroft, *Introduction to automata theory, languages, and computation*. Pearson Education India, 2008.

[39] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[40] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International Conference on Machine Learning*, 2016, pp. 2091–2100.

[41] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, "OpenNMT: Open-Source Toolkit for Neural Machine Translation," *ArXiv e-prints*.

[42] 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.

[43] D. R. Reddy *et al.*, "Speech understanding systems: A summary of results of the five-year research effort," *Department of Computer Science. Camegie-Mell University, Pittsburgh, PA*, 1977.

[44] 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.

[45] "Gerrit Code Review Database," https://www.gerritcodereview.com/, 2018, [Online; accessed 18-August-2018].

[46] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 200–20 010.

[47] S. Kommrusch, T. Barollet, and L.-N. Pouchet, "Equivalence of dataflow graphs via rewrite rules using a graph-to-sequence neural model," *arXiv preprint arXiv:2002.06799*, 2020.

[48] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 1157–1168.

[49] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[50] W. J. Masek and M. S. Paterson, "A faster algorithm computing string edit distances," *Journal of Computer and System sciences*, vol. 20, no. 1, pp. 18–31, 1980.

[51] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[52] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*. IEEE, 2017, pp. 648–659.

[53] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 298–309.

[54] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 24–36.

[55] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 802–811.

[56] S. Saha, R. k. Saha, and M. r. Prasad, "Harnessing evolution for multi-hunk program repair," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 13–24.

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

[58] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: bug detection with n-gram language models," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 708–719.

[59] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

[60] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 87–98.

[61] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 933–944.

[62] 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.

[63] C. Maddison and D. Tarlow, "Structured generative models of natural source code," in *International Conference on Machine Learning*, 2014, pp. 649–657.

[64] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1. IEEE, 2015, pp. 858–868.

[65] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, p. 40, 2019.

[66] S. Karaivanov, V. Raychev, and M. Vechev, "Phrase-based statistical translation of programming languages," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM, 2014, pp. 173–184.

[67] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Divide-and-conquer approach for multi-phase statistical migration for source code," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 585–596.

[68] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical learning approach for mining API usage mappings for code migration," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 457–468.

[69] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," *arXiv preprint arXiv:1802.03691*, 2018.

[70] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "DeepFix: Fixing common C language errors by deep learning." in *AAAI*, 2017, pp. 1345–1351.

[71] 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*, 2019, pp. 6563–6573.

[72] 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.

[73] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu, "Commit message generation for source code changes." in *IJCAI*, 2019, pp. 3975–3981.

[74] 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.

[75] I. Polosukhin and A. Skidanov, "Neural program search: Solving programming tasks from description and examples," *arXiv preprint arXiv:1802.04335*, 2018.

[76] M. Allamanis and C. Sutton, "Mining idioms from source code," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 472–483.

[77] M. Allamanis, E. T. Barr, C. Bird, P. Devanbu, M. Marron, and C. Sutton, "Mining semantic loop idioms," *IEEE Transactions on Software Engineering*, vol. 44, no. 7, pp. 651–668, 2018.

[78] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov, "Generative code modeling with graphs," *arXiv preprint arXiv:1805.08490*, 2018.

[79] P. Yin, G. Neubig, M. Allamanis, M. Brockschmidt, and A. L. Gaunt, "Learning to represent edits," *arXiv preprint arXiv:1810.13337*, 2018.

[80] 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," *arXiv preprint arXiv:1911.01205*, 2019.

[81] 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.

[82] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, "Minthint: Automated synthesis of repair hints," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 266–276.

[83] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 3–13.

[84] P. Liu, O. Tripp, and X. Zhang, "Flint: fixing linearizability violations," in *ACM SIGPLAN Notices*, vol. 49, no. 10. ACM, 2014, pp. 543–560.

[85] F. Logozzo and T. Ball, "Modular and verified automatic program repair," in *ACM SIGPLAN Notices*, vol. 47, no. 10. ACM, 2012, pp. 133–146.

[86] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, p. 54, 2012.

[87] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 166–178.

[88] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 1–11.

[89] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 772–781.

[90] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*. ACM, 2016, pp. 691–701.

[91] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*. IEEE, 2008, pp. 162–168.

[92] X.-B. D. Le, D. Lo, and C. Le Goues, "History driven automated program repair," 2016.

[93] M. Boshernitsan, S. L. Graham, and M. A. Hearst, "Aligning development tools with the way programmers think about code changes," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2007, pp. 567–576.

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

[95] Y. Tian and B. Ray, "Automatically diagnosing and repairing error handling bugs in c," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 752–762.

[96] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: suggesting solutions to error messages," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2010, pp. 1019–1028.

[97] V. Murali, L. Qi, S. Chaudhuri, and C. Jermaine, "Neural sketch learning for conditional program generation," *arXiv preprint arXiv:1703.05698*, 2017.

[98] P. Flener, *Logic program synthesis from incomplete information*. Springer Science & Business Media, 2012, vol. 295.

[99] T. Lutellier, L. Pang, V. H. Pham, M. Wei, and L. Tan, "Encore: Ensemble learning using convolution neural machine translation for automatic program repair," *arXiv preprint arXiv:1906.08691*, 2019.