Learning to Generate Corrective Patches Using Neural Machine Translation

Hideaki Hata, *Member, IEEE*, Emad Shihab, *Senior Member, IEEE*, and Graham Neubig

Abstract—Bug fixing is generally a manually-intensive task. However, recent work has proposed the idea of automated program repair, which aims to repair (at least a subset of) bugs in different ways such as code mutation, etc. Following in the same line of work as automated bug repair, in this paper we aim to leverage past fixes to propose fixes of current/future bugs. Specifically, we propose Ratchet, a corrective patch generation system using neural machine translation. By learning corresponding pre-correction and post-correction code in past fixes with a neural sequence-to-sequence model, Ratchet is able to generate a fix code for a given bug-prone code query. We perform an empirical study with five open source projects, namely Ambari, Camel, Hadoop, Jetty and Wicket, to evaluate the effectiveness of Ratchet. Our findings show that Ratchet can generate syntactically valid statements 98.7% of the time, and achieve an F1-measure between 0.29 – 0.83 with respect to the actual fixes adopted in the code base. In addition, we perform a qualitative validation using 20 participants to see whether the generated statements can be helpful in correcting bugs. Our survey showed that Ratchet's output was considered to be helpful in fixing the bugs on many occasions, even if fix was not 100% correct.

Index Terms—patch generation, corrective patches, neural machine translation, change reuse.

1 Introduction

Most software bug fixing tasks are manual and tedious. Recently, a number of techniques related to automated program repair have been proposed to help automate and reduce the burden of some of these tasks [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25]. These systems are also seeing practical use. For example, Facebook has announced that they started applying a system of automated program repair called SapFix in their large-scale products [26].

However, there are limitations in current approaches to automated program repair. First, there is a risk of overfitting to the training set (and breaking under tested functionality) in patch generation, especially generated tests tends to lead overfitting compared to human-generated, requirements-based test suites [27]. Second, correct patches may not exist in the search space, or correct patches cannot be generated because the search space is huge [28], [29]. Several studies address this search space issue by making use of existing human-written patches [30], [31], [32], [33], [34], [35], [36], [37], but those generated patches need to be validated with test suites. Therefore, investigating techniques that assist in the generation of patches without the need for tests, etc. are needed. Instead of exploring fix ingredients in the search space (search-based), we study the possibility of learning fix ingredients from past fixes (learning-based).

Recently, Neural Machine Translation (NMT) has been proposed and showed promising results in various areas including not only translation between natural languages (such as English and Japanese), but also other NLP tasks such as speech recognition [38], natural language parsing [39], and text summarization [40]. Similar techniques have been applied to code-related tasks [41], [42], [43], [44], [45], [46], [47]. The notable success of NMT in such a wide variety of tasks can be attributed to several traits: (1) It is an end-to-end machine learning framework that can be learned effectively from large data – if we have a large enough data source it is able to learn even complicated tasks such as translation in an effective way; (2) Unlike previous models for translation such as phrase-based translation [48] (which has also been used in code-related tasks such as language porting [49] and pseudo-code generation [50]), NMT is able to take a holistic look at the entire input and make global decisions about which words or tokens to output. In particular, for bug fixing we posit this holistic view of the entire hunk of code we attempt to fix is important, and thus focus on approaches using NMT in this work.

Hence, in this paper, we propose Ratchet, a NMT-based technique that generates bug fixes based on prior bug-and-fix examples. To evaluate the effectiveness of the technique, we perform an empirical study with five large software projects, namely Ambari, Camel, Hadoop, Jetty and Wicket. We use the pattern-based patch suggestion inspired by the Plastic Surgery work [51], as a comparison baseline and examine the effectiveness of our NMT-based technique. In particular, we quantify the number of cases where our NMT-based technique is able to generate a valid fix and how accurate the generated fixes are. Our findings showed that Ratchet is able to generate a valid statements in 98.7% of the cases and achieves an F1 measure between 0.29 – 0.83 with respect to the actual fixes adopted in the code base. For all five projects, Ratchet was able to either outperform or

- *H. Hata is with Division of Information Science, Nara Institute of Science and Technology, Nara, 6300192, Japan. E-mail: hata@is.naist.jp*
- *E. Shihab is with Department of Computer Science and Software Engineering, Concordia University, Montreal, QC H3G 1M8, Canada. E-mail: eshihab@cse.concordia.ca*
- *G. Neubig is with Language Technology Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3891, USA. E-mail: gneubig@cs.cmu.edu*

Manuscript received April 19, 2005; revised August 26, 2015.

perform as well as the baseline.

In addition to the quantitative validation, we also performed a survey with 20 participants to see whether the generated statements can help in correcting a bug (even if they were not 100% identical to the fix). Our findings through a participant survey show that the fixes generated by Ratchet are very helpful, even if they were not fully correct (although the correct fixes were most helpful).

There are several recent studies on techniques to generate patches without test cases, which differ from our approach: *inductive programming* for program synthesis making used of historical change patterns [52], additive program transformations using *separation logic* to identify and repair the lack of certain operations on heap issues [53], and learning fix patterns of FindBugs violations using *convolutional neural networks* [54]. Similar to our approach, these proposals have learning aspects to generate patches without test cases. Major differences are specific targets (heap properties [53] and static analysis tool violations [54]) and/or specific patterns to be learned (specified constraints [52] and manual fix specifications [53]), while Ratchet learns any statement-level changes in a general NMT framework. Although limiting to specific targets and patterns could be effective for the targeted domains, our approach is able to target daily bug fixing activities.

Our approach can be thought of as a method for "learning-based automated code changes" instead of one of automated program repair per se. Although the setting on automated program repair is expensive, especially for validation [21], our NMT approach can work lightly for usual repetitive maintenance activities. As automated program repair research is recommended to focus on difficult and important bugs [55], research on learning-based automated code changes could support repetitive and similar bug fixing tasks by learning common corrective maintenance activities. We expect that our approach can be integrated in daily maintenance activities. Ratchet can recommend generated patches to local code before committing to repositories and to submitted code for reviewing. While it could work in human-involved maintenance processes, we consider our approach is not necessarily an end-to-end bug fixing solution by assessing the correctness of generated patches.

The rest of the paper is organized as follows. Section 2 presents relevant terminology. Section 3 provides background about NMT. Section 4 details our approach. Section 5 sets up our experiments, discussing their design and the data used. Section 6 presents our results and Section 7 discusses the generality and some challenges facing NMT-based solutions. Related work is presented and contrasted in Section 8 and Section 9 concludes the paper.

2 Terminology

We use the term, *change hunk*, similar to the previous study by Ray et al. [56]. *A change hunk is a list of program statements deleted and added contiguously.* In a single commit to a code repository, typically there are multiple change regions in multiple files. Even in a single file, there can be multiple change regions. Those changed regions can be identified with *diff*. Although the previous study by Ray et al. included unchanged statements in a change hunk [56], we do not include them. We call deleted and added statements *pre-correction* and *post-correction* statements respectively. In Listing 1, the red statement is a pre-correction statement and the green statement is a corresponding post-correction statement, and these associated two statements are considered to be a change hunk.

Listing 1. An example of a change hunk in bug fixing.

```
Commit: 44074f6ae03031ab046b1886790fc31e66e2d74e
Author: Willem Ning Jiang
Date: Sat Jun 9 09:24:15 2012 +0000
Message: CAMEL-5348 fix the issue of Uptime

  uptime /= 24;
  long days = (long) uptime;
- long hours = (long) ((uptime - days) * 60);
+ long hours = (long) ((uptime - days) * 24);
  String s = fmtI.format(days)
                 + (days > 1 ? " days" : " day");
```

In this study, we are interested in learning transforming patterns between corresponding pre-correction and post-correction statements. Thus, we ignore change hunks that only contain deleted or added statements. All change hunks studied in this paper are pairs of pre-correction and post-correction statements.

3 Background

Neural machine translation (NMT), also called neural sequence-to-sequence models (seq2seq) [57], [58], [59] is a method for converting one input sequence $\boldsymbol{x}$ into another output sequence $\boldsymbol{y}$ using neural networks. As the name suggests, the method was first conceived for and tested on machine translation; for converting one natural language (e.g. English) into another (e.g. French). However, because these methods can work on essentially any problem of converting one sequence into another, they have also been applied to a wide variety of other tasks such as speech recognition [38], natural language parsing [39], and text summarization [40]. They have also seen applications to software for generation of natural language comments from code [41], generation of code from natural language [42], [43], [44], generation of API sequences [45], and suggesting fixes to learner code in programming MOOCs [46], [47].

In this section we briefly overview neural networks, then explain NMT in detail.

3.1 Neural Networks

Neural networks [60], put simply, are a complicated function that is composed of simpler component parts that each have parameters that control their behavior. One common example of such a function is the simple multi-layer calculation below, which converts an input vector $\boldsymbol{x}$ into an output vector $\boldsymbol{y}$:

$$\begin{aligned} \boldsymbol{h} &= tanh(W_1\boldsymbol{x} + b_1) \\ \boldsymbol{y} &= W_2\boldsymbol{h} + b_2. \end{aligned} \tag{1}$$

Here, W_1 and W_2 are parameter matrices, and b_1 and b_2 are parameter vectors (called *bias* vectors). Importantly, the vector $\boldsymbol{h}$ is a *hidden* layer of the neural network, which results from multiplying W_1, adding b_1, then taking the hyperbolic tangent with respect to the input. This hidden layer plays an essential role in neural networks, as it allows the network to automatically discover features of the input that may be useful in predicting $\boldsymbol{y}$.[1]

Because neural networks have parameters (W_1, b_1, etc.) that specify their behavior, it is necessary to learn these parameters from training data. In general, we do so by calculating how well we do in predicting the correct answer $\boldsymbol{y}'$ provided by the training data, and modify the parameters to increase our prediction accuracy. Formally, we do so by calculating a *loss function* $\ell(\boldsymbol{y}, \boldsymbol{y}')$ which will (generally) be 0 if we predict perfectly, and higher if we're not doing a good job at prediction. We then take the derivative of this loss function with respect to the parameters, e.g. $\frac{\partial \ell(\boldsymbol{y},\boldsymbol{y}')}{\partial W_1}$, and move the parameters in the direction to reduce the loss function, e.g.

$$W_1 \leftarrow W_1 - \alpha \frac{\partial \ell(\boldsymbol{y}, \boldsymbol{y}')}{\partial W_1}, \tag{2}$$

where α is a learning rate that controls how big of a step we take after every update.

The main difficulty here is that we must calculate derivatives $\frac{\partial \ell(\boldsymbol{y},\boldsymbol{y}')}{\partial W_1}$. Even for a relatively simple function such as the one in (1), calculating the derivative by hand can be cumbersome. Fortunately, this problem can be solved through a process of *back-propagation* (or *auto-differentiation*), which calculates the derivative of the whole complicated function by successively calculating derivatives of the smaller functions and multiplying them together using the chain rule [62]. Thus, it becomes possible to train arbitrarily complicated functions, as long as they are composed of simple component parts that can be differentiated, and a number of software libraries such as TensorFlow [63] and DyNet [64] make it possible to easily do so within applications.

3.2 Neural Machine Translation

NMT is an example of applying a complicated function learnable by neural nets and using it to solve a complicated problem: translation. To generate an output $\boldsymbol{y}$ (e.g. corrected hunk of code) given an input $\boldsymbol{x}$, these models incrementally generate each token in the output $y_1, y_2, \ldots, y_{|\boldsymbol{y}|}$ one at a time. For example, if our output is "`return this . index`", the model would first predict and generate "`return`", then "`this`", then "`.`", etc. This is done in a probabilistic way by calculating the probability of the first token of the output given the input $P(y_1 \mid \boldsymbol{x})$, outputting the token in the vocabulary that maximizes this probability, then calculating the probability of the second token given the first token and the snippet $P(y_2 \mid \boldsymbol{x}, y_1)$ and similarly outputting the word with the highest probability, etc. When training the model, we already know a particular output $\boldsymbol{y}$ and want to calculate its probability given a particular snippet $\boldsymbol{x}$ so we can update the parameters based on the derivatives of this probability. To do so, we simply multiply these probabilities together using the chain rule as follows:

$$P(\boldsymbol{y} \mid \boldsymbol{x}) = P(y_1 \mid \boldsymbol{x}) P(y_2 \mid \boldsymbol{x}, y_1) P(y_3 \mid \boldsymbol{x}, y_1, y_2) \ldots \tag{3}$$

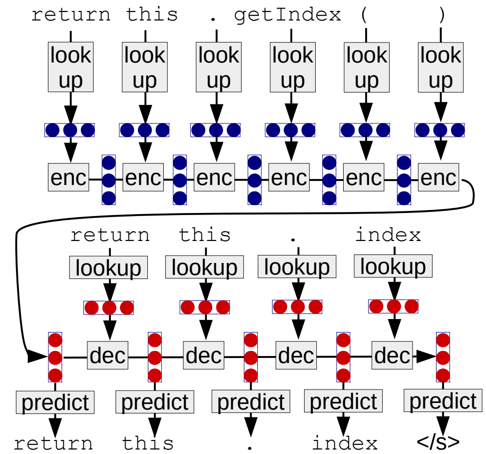


Fig. 1. An example of NMT encoder-decoder framework used in translation.

So how do NMT models calculate this probability? We will explain a basic outline of a basic model called the *encoder-decoder model* [58], and refer readers to references for details [58], [59], [65]. The encoder-decoder model, as shown in Figure 1 works in two stages: first it *encodes* the input (in this case $\boldsymbol{x}$) into a *hidden vector* of continuous numbers $\boldsymbol{h}_x$ using an encoding function

$$\boldsymbol{h}_{x,|\boldsymbol{x}|} = \text{encode}(\boldsymbol{x}). \tag{4}$$

This function generally works in two steps: looking up a vector of numbers representing each token (often called "word embeddings" or "word vectors"), then incrementally adding information about these embeddings one token at a time using a particular variety of network called a *recurrent neural network* (RNN). To take the specific example shown in the figure, at the first time step, we would look up an embedding vector for the first token "`return`", $\boldsymbol{e}_1 = \boldsymbol{e}_{\texttt{return}}$ and then perform a calculation such as the one below to calculate the hidden vector for the first time step:

$$\boldsymbol{h}_{x,1} = \tanh(W_{\text{enc,e}} \boldsymbol{e}_1 + b_{\text{enc}}), \tag{5}$$

where $W_{\text{enc,e}}$ and b_{enc} are a matrix and vector that are parameters of the model, and $tanh(\cdot)$ is the hyperbolic tangent function used to "squish" the values to be between -1 and 1.[2] In the next time step, we would do the same for the symbol ".", using its embedding $\boldsymbol{e}_2 = \boldsymbol{e}_{.}$, and in the calculation from the second step onward we also use the result of the previous calculation (in this case $\boldsymbol{h}_{x,2}$):

$$\boldsymbol{h}_{x,2} = \tanh(W_{\text{enc,h}} \boldsymbol{h}_{x,1} + W_{\text{enc,e}} \boldsymbol{e}_2 + b_{\text{enc}}). \tag{6}$$

By using the hidden vector from the previous time step, the RNN is able to "remember" features of the previously occurring tokens within this vector, and by repeating this process until the end of the input sequence, it (theoretically) has the ability to remember the entire content of the input within this vector.

Once we have encoded the entire source input, we can then use this encoded vector to predict the first token of the

1. In fact, by adding this hidden layer, a simple function such as the above is able to accurately perform any prediction task given a large enough $\boldsymbol{h}$ and enough training data, and thus neural networks are known as *universal function approximators* [61].

2. This represents a simple recurrent neural network, but in our actual model we use a more sophisticated version of encoding function called "long short-term memory" (LSTM), which performs better on long sequences [66].

output. This is generally done by defining the first hidden vector for the output $\boldsymbol{h}_{y,0}$ to be equal to the final vector of the input $\boldsymbol{h}_{x,|\boldsymbol{x}|}$, then multiplying it with another weight vector used for prediction to calculate a score $\boldsymbol{g}$ for each token in the output vocabulary:

$$\boldsymbol{g}_1 = W_{\text{pred}}\boldsymbol{h}_{y,0} + \boldsymbol{b}_{\text{pred}}. \quad (7)$$

We then predict the actual probability of the first token in the output statement, for example "`return`", by using the *softmax* function, which exponentiates all of the scores in the output vocabulary and then normalizes these scores so that they add to one:

$$P(y_1 = \text{“}\texttt{return}\text{”}) = \frac{\exp(g_{\text{return}})}{\sum_{\tilde{g}} \exp(\tilde{g})}. \quad (8)$$

We then calculate a new hidden vector given this input:

$$\boldsymbol{h}_{y,1}\text{encode}(y_1 = \text{“}\texttt{return}\text{”}, \boldsymbol{h}_{y,0}). \quad (9)$$

We continue this process recursively until we output a special "end of hunk" symbol "$\langle$/s$\rangle$".

Why NMT models?: As mentioned briefly in the intro, NMT models are well-suited to the task of automatic patch generation for a number of reasons. First, they are an end-to-end probabilistic model that can be trained from parallel datasets of pre- and post-correction code without extra human intervention, making them easy to apply to new datasets or software projects. Second, they are powerful models that can learn correspondences on a variety of levels; from simple phenomena such as direct token-by-token matches, to soft paraphrases [67], to weak correspondences between keywords and large documents for information retrieval [68]. Finally, they have demonstrated success in a number of code related tasks as iterated at the beginning of this section, which indicates that they should be useful as part of bug fixing algorithm as well.

Attention: In addition, we use a NMT model with this basic architecture, with the addition of a feature called *attention*, which, put simply, allows the model to "focus" on particular tokens in the input $\boldsymbol{x}$ when generating the output $\boldsymbol{y}$ [65], [69]. Mathematically, this corresponds to calculating an "attention vector" $\boldsymbol{a}_j$, given the input hidden vectors $\boldsymbol{h}_x$ and the current output hidden vector $\boldsymbol{h}_{y,j}$. This vector consists of values between zero and one, one value for each word in the input, with values closer to one indicating that the model is choosing to focus more on that particular word. Finally, these values are used to calculate a "context vector"

$$\boldsymbol{c}_j = \sum_{i=1}^{|\boldsymbol{x}|} \alpha_{j,i}\boldsymbol{h}_{x,i}, \quad (10)$$

which is used as additional information when calculating score $\boldsymbol{g}_j$. Attention is particularly useful when there are many token-to-token correspondences between the input and output, which we expect to be the case for our patch generation task, where the input and output code are likely to be very similar. This attention model can be further augmented to allow for exact copies of tokens [70], or be used to incorporate a dictionary of common token-to-token correspondences (copies or replacements) [71]. In our model, we use the latter, which allows us to both capture the fact that tokens are frequently copied between pre- and post-correction code, and also the fact that some replacements will be particularly common (e.g. `loadBalancerType` to `setLoadBalancerType`). This dictionary is automatically inferred from our training data by running the `fast_align` toolkit[3], which can automatically learn such a dictionary from parallel data using probabilistic models [72].

Implementation details: As a specific implementation of the NMT techniques listed above, we use the lamtram toolkit [73]. For reproducibility, we briefly list the parameters below, and interested readers can refer to the references for detail. As our model we use an encoder-decoder model with multi-layer perceptron attention [65] and input feeding [69], with encoders and decoders using a single layer of 512 LSTM cells [66]. We use the Adam optimizer [74] with a learning rate of 0.001 and minibatch size of 2048 words, and decay the learning rate every time the development loss increases. To prevent overfitting, we use a dropout rate of 0.5 [75]. To generate our outputs, we perform beam search with a beam size of 10.

4 Approach

The idea of corrective patch generation using NMT considers code changes as translation from pre-correction code to post-correction code. Figure 2 provides an overview of our system, Ratchet, which consists of two main parts: creating the training corpora, and generating patches using the trained model. In this paper, we target Java source code and focus on changes within Java methods. Particularly, the granularity of code we target is a statement similar to the previous study [51]. Main focus in this study is preparing appropriate data for a NMT model to learn. To this aim, we build a system to collect fine-grained code change and try ignoring noisy data.

4.1 Extracting Change Hunks from Code Repositories

In order to create our training corpora, we start by extracting pre- and post-correction statements using a sequence of steps. We detail each of these steps in the following text:

Preparing Historage for method-level histories. Since the software repositories store the code modifications at the commit level, our first step is to transform these commits into method-level modifications. To do so, we convert the existing code repositories to historage repositories [76]. Historage creates a new repository that stores all methods in the logs of the original repository as individual Git objects. In essence, historage is a Git repository that allows us to operate any Git commands as usual.[4]

Collecting the modified methods. We use the command `git log --diff-filter=M` on the historage repositories to collect all modified methods in the entire history. The option `--diff-filter=M` will provide only modified (M) files, which are methods in historage repositories. Since

3. https://github.com/clab/fast_align

4. We used a tool, called kenja [77] (https://github.com/niyaton/kenja) to prepare the historage repositories. Converted historage repositories are now hosted in Codosseum Web service: http://codosseum.naist.jp/, which is previously presented as Kataribe [77], [78].

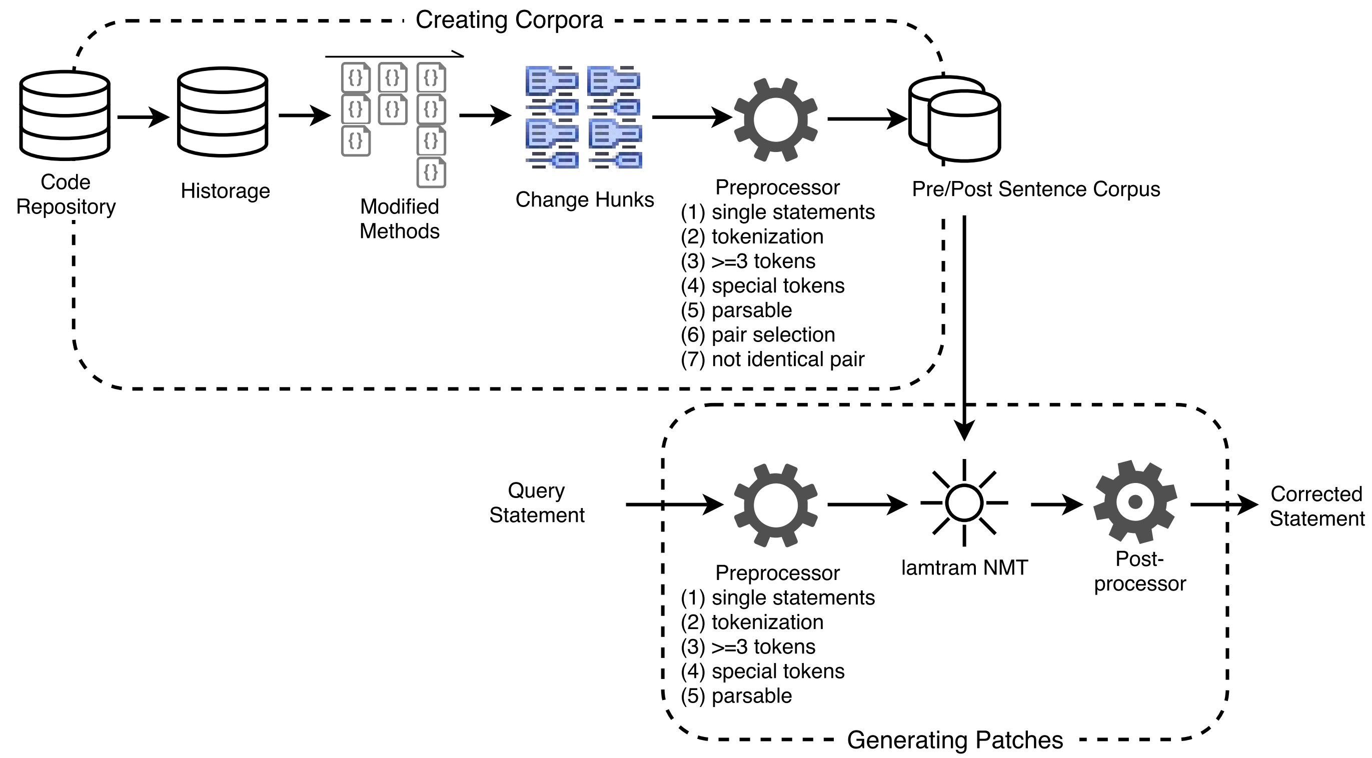


Fig. 2. Overview of Ratchet, an NMT-based corrective patch generation system.

we are interested in training our model on pre- and post-correction statements, we only consider methods that modify code, i.e., not methods that are newly created or completely deleted.

Identifying change hunks. As stated in Section 2, a change hunk is a pair of pre-correction and post-correction statements. We identify these change hunks from the outputs of the `git diff`. Since we assume pre-correction statements have been corrected to post-correction statements, we need to identify the corresponding line pairs appropriately.[5]

4.2 Preprocessing the Statement Corpora

Before storing the statement pairs as pre-correction and post-correction statement corpora, we perform the following preprocessing steps. As seen in Figure 2, the same processes will be applied to query statements except for the step (6) and (7), which are needed only for creating the corpora.

(1) Limit to single-statement changes and single-statement queries. In this study, we only consider single-statement (one-line) changes. We do so for the following three reasons. First, previous studies showed that most reusable code is found at the single-statement level [51], [80]. Second, it is difficult to treat multiple statement changes (one-to-many, many-to-one, and many-to-many statement changes) for identifying pairs. Those multiple statement changes can have inappropriate corresponding statements. For example, if there exists one pre-correction statement and two post-correction statements in one change hunk, this change can be a single-statement change and one independent statement insertion. If we consider these statements one pair, the independently inserted statement can be noise in the training data. Third, it is difficult to manage past histories associated with multiple statements. Using the command `git blame` on historage, we identify commits on which deleted lines initially appeared. In general, multiple statements can have different past histories, which makes it difficult to treat those multiple statements as one unit. For all statement pairs, we collect past history information including the original commit, changed year and deleted year, to be used for our experiments. Although we apply this filtering, we found that single-statement changes are the majority in our change hunks (as we show later in Figure 3 and Table 2).

(2) Tokenize statements. Since the NMT model requires separate tokens as input, we use the StreamTokenizer to tokenize the Java statements.

(3) Remove statement pairs or statement queries with less than three tokens. We remove statements that have very few tokens (i.e., less than 3) since they are less meaningful. Our observations indication that most such lines only contain opening or closing parenthesis.

(4) Replace the contents of method arguments with a special token. From our many trials, we realized that a wide variety of the contents of method arguments make it difficult to generate corresponding contents. This is because sometimes method argument contents include tokens that rarely appear. We replace method and array arguments with a special token, `arg` and `val`, respectively.

(5) Filter unparseable statement pairs and queries.

5. The options `--histogram`, `--unified=0`, `--ignore-space-change`, `--ignore-all-space`, and `--ignore-blank-lines` are used to apply an advanced diff algorithm, ignore unchanged statements, and ignore trivial changes. An empirical study reported that the histogram algorithm is better than the default diff algorithm in Git [79].

There exist incomplete statements in our collected statements, e.g., when there is a long statement that is written across two lines, and only one line is changed. To remove these incomplete Java statements, we put each statement in a dummy method of a dummy class, and try parsing the class to get an AST using JavaParser.[6] If we fail to parse classes with either pre- or post-correction statements, we filter out the failed statement pairs

(6) Select post-correction statements from multiple candidates. This step is performed to address *the nature of sequential order* in documents. After collecting all pre- and post-correction statements from the entire history of a code repository, we can have statement pairs that have the same pre-correction statements but different post-correction statements. In order to allow the NMT models to effectively extract relationships or patterns, we chose only one post-correction statement for one pre-correction statement, and remove all other post-correction statements. The idea behind this selection is that it is better to learn from recently and frequently appearing statements. Given a pre-correction statement, we obtain post-correction statements that appeared in the most recent year. Then, from those newer statements, we select statements that most frequently appeared in the entire history. If we cannot break ties, we select the first statement in alphabetical order to make the process deterministic.

(7) Remove identical pre- and post-correction statements. After the above processes, there exist pairs of identical pre- and post-correction statements. For example, statement pairs from changes only within method arguments, and white space changes. We remove those statement pairs.

4.3 Post-Processing

Since we replace the contents of method arguments and replace it with a special token, the NMT model does not generate method arguments. However we expect that the method arguments of a query statement can be reused in the generated statement. Therefore we prepare the following heuristics for new method arguments.

- Methods that have the same name will have the same method arguments.
- For chained method calls, arguments are assigned in the same order.
- If no method argument content is left in a query statement, leave the remaining method call arguments empty.

The lamtram toolkit provides scores associated with generated statements with the logarithm of a posteriori probability of output E given input F as $logP(E|F)$. Those scores can be considered as confidences of the results. We empirically determine thresholds and ignore the generated statements with low scores. In addition, we can also ignore invalid generated statements that cannot be parsed.

5 Experimental Setup

In this section, we discuss our dataset and the design of our experiment. Particularly, we are interested in examining the viability of our approach in generating bug-fixing statements. To do so, we need to identify bug-fixing statement pairs. We discuss the tool used to identify the bug-inducing and bug-fixing commits that are used to determine our bug-fix statement pairs. Then, we provide descriptive statistics about the studied datasets.

6. JavaParser: http://javaparser.org/

TABLE 1
Descriptive statistics of the studied systems. The number of Java files and methods are from the latest snapshots.

Project	Period	# of Commits	# of Files	# of Methods
ambari	Aug-11 to Apr-17	14,042	2,719	29,212
camel	Mar-07 to Jun-17	28,668	16,889	92,839
hadoop	May-09 to Oct-14	8,323	5,696	21,292
jetty	Mar-09 to Apr-16	14,167	2,668	21,172
wicket	Sep-04 to Jun-17	19,960	5,039	16,049

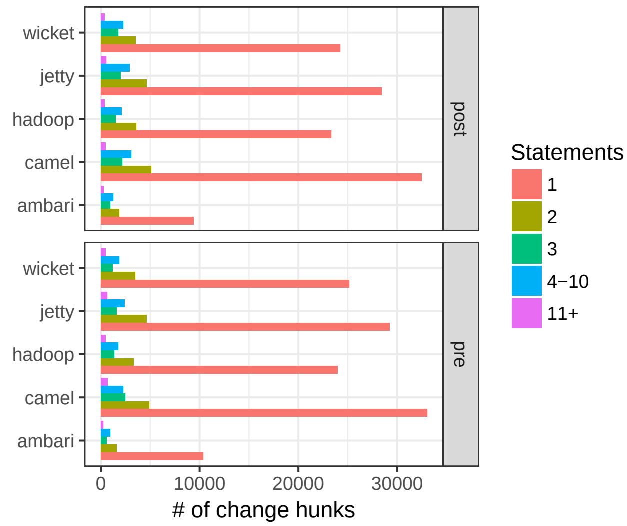


Fig. 3. The number of change hunks with different numbers of pre- and post-correction statements in the entire periods.

5.1 Subject Projects

To perform our case study, we study five projects, namely Apache Ambari, Apache Camel, Apache Hadoop, Eclipse Jetty and Apache Wicket. We chose to study these five projects since they have long development histories and are large projects that contain many commits. Table 1 shows the period considered, the number of commits, files and methods in our dataset.

Figure 3 shows the distribution of the number of pre- and post-correction statements in all change hunks (counted separately). We find that most of changes are single statements in either insertion, deletion, or modification. Multi-statement changes are not frequent. Table 2 shows the number of all change hunks and the number of change hunks that are derived from single-statement changes. We see from the table that approximately 62 – 68% of the changes are single-statement changes. Since we investigated changes per methods using historage repositories [76], we could divide large modifications in files [81], [82] to fine-grained changes,

TABLE 2
Statistics of change hunks in the entire period.

Project	# of All hunks	#, (%) of Hunks with single-statement pairs	
ambari	13,701	8,565	(62.5%)
camel	43,237	28,672	(66.3%)
hadoop	30,806	21,049	(68.3%)
jetty	38,443	25,517	(66.4%)
wicket	32,132	21,926	(68.2%)

which results in high rations of single-statement changes. These ratios are encouraging for Ratchet, which is limited to single-statement changes.

5.2 Experimental Design

From the collected pre- and post-correction statements, we prepare the training data (Table 3) and testing data (Table 4). Considering the number of statements, we set the testing year for each project as shown in Table 4. All statement pairs in each testing year are used as testing data, which means we chose statement pairs whose pre-correction statements are created in the testing year and changed to the corresponding post-correction statements in the same testing year. All years before the testing year are considered as training periods. In each training period, the numbers of statement pairs, whose pre-correction statements are changed to post-correction statements in the training period, are shown in Table 3.

This experimental design can be regarded as a simulation of generating corrected statements only by learning past histories when new statements are created and they will be modified soon (in the same year). If this works, we can prevent recurring or similar issues before being inserted into the code, or even when the code is being edited. For this purpose, we prepare the training and testing data by considering chronological order instead of random partitioning. For the risk of increasing unseen changes in the training data, we limit the testing year to one year.

5.3 Data Preparation

Table 3 details the impact of the various preprocessing steps on our approach. The before filtering row shows the number of all single-statement change pairs. The <3 tokens row shows the effect of removing statements that have less than 3 tokens. Then we remove the unparsable statements in both, pre-correction and post-correction statements. The final step removes identical statement pairs in the pre- and post-correction statements. The last row shows the final number of statements used in our study.

In addition, we perform specific processing for the training and testing data, which we detail below:

Replacing rare tokens in the training data. From the processed statement pairs, we prepare pre-correction statement corpus and post-correction statement corpus. For each corpus, tokens that appear only once are replaced with ⟨unk⟩, which is a common way to handle unknown tokens [59]. This script is available in the lamtram toolkit.[7]

Categorization of testing data. When testing our approach, we call the pre-correction statements in the testing data as *queries*. On the other hand, we call the post-correction statements as *references*.

When we evaluate our approach, we separate the testing data with their characteristics. First, all statement pairs in the testing data are divided into bug-fix statement pairs and non-bug-fix statement pairs. This classification procedure is presented in Section 5.4. Then both classes of statement pairs are categorized into three:

- NU: **No unknown.** There are no unknown tokens in a statement pair. All tokens in a query statement appear in the pre-correction statement from the training data corpus, and all tokens in a reference statement appear in the post-correction statement of the training data corpus.
- UQ: **Unknown in query.** One or more token(s) in the query statement do not appear in the pre-correction statement corpus. In other words, there are unknown tokens in the query.
- UR: **Unknown in reference.** Although there is no unknown token in the query statement, there are one or more unknown token(s) in the reference, i.e., in the corresponding post-correction statement.

We categorize the statements as shown above to know which data can be used in our experiments. This is particularly important since the trained NMT models have not seen *unknown* tokens during training, addressing queries in UQ or UR is very difficult. In fact, it is impossible for our model to generate statements that are the exact same as the references for the UR category.[8]

Table 4 shows the number of statement pairs for these categories of bug-fixing and non-bug fixing classes. As can be seen from the Table 4, the majority of the training data's statements fall in the UQ category (except for the Jetty project). On the other hand, the good news is that statements in the UR category are the least. We evaluate our approach using statements in the NU category.

The dataset is available online.[9]

5.4 Identifying Bug-Fixing Statements

We collect bug-fixing statement pairs by identifying the pairs of bug-inducing and bug-fixing commits. To obtain these commits, we use `Commit.guru` [84], a tool that analyzes and provides change level analytics.[10] For full details about commit.guru, we point the reader to the paper by Rosen *et al.* [84], however, here we describe the relevant details for our paper. Commit.guru takes as input a Git repository address, an original code repository in this study, and provides data for all commits of the project. It applies

7. lamtram: https://github.com/neubig/lamtram
8. These errors could potentially be alleviated by incorporating a mechanism to copy inputs from the source [70] or generate tokens in several sub-token parts [83].
9. Train and test dataset for this study: https://github.com/hideakihata/NMTbasedCorrectivePatchGenerationDataset
10. Commit Guru: http://commit.guru

TABLE 3
Filtering results for training data.

	ambari		camel		hadoop		jetty		wicket	
train period	2011-2013		2007-2013		2009-2012		2009-2013		2004-2014	
#, (%) of statement pairs										
before filtering	4,253	(100%)	25,562	(100%)	11,952	(100%)	21,278	(100%)	21,268	(100%)
(3) <3 tokens	70	(1.7%)	285	(1.1%)	69	(0.6%)	281	(1.3%)	127	(0.6%)
(5) not parsable	261	(6.2%)	2,924	(11.4%)	1,239	(10.4%)	2,023	(9.5%)	3,092	(14.5%)
(6) lost candidates	88	(2.1%)	1,489	(5.8%)	384	(3.2%)	1,186	(5.6%)	1,324	(6.2%)
(7) identical	2,353	(55.3%)	11,346	(44.4%)	5,812	(48.6%)	7,755	(36.4%)	7,068	(33.2$)
final statement pairs	1,481	(34.8%)	9,518	(37.2%)	4,448	(37.2%)	10,033	(47.2%)	9,657	(45.4%)

TABLE 4
Summary of testing data.

	ambari	camel	hadoop	jettty	wicket
test year	2014	2014	2013	2014	2015
NU	6	42	29	149	7
UQ	30	54	167	31	16
UR	0	8	15	13	1

TABLE 5
Number of the generated valid statements for *buggy* queries.

ambari	camel	hadoop	jetty	wicket
6/6 (100%)	42/42 (100%)	28/29 (97%)	147/149 (99%)	7/7 (100%)

the SZZ algorithm [85] to identify bug-inducing commits and their associated bug-fixing commits. In addition, Commit.guru provides a number of change level metrics related to the size of the change, the history of the files changed, the diffusion of the change and the experience of the developers making the modification.

As mentioned earlier in step (1) of preprocessing (Section 4.1), we have meta information of statements including the original commits of post-correction statements and the original commits of pre-correction statements. We consider a pair of statements bug fixing if and only if a pre-correction statement is created in a bug-inducing commit and an post-correction statement is created in the associated bug-fixing commit.[11] The other statement pairs are treated as non-bug-fix statements. We do not distinguish the types of the training data, that is, bug-fix or non-bug-fix. This is because we prefer to increase the training data available to the model and make the model learn from all varieties of changes.

6 Evaluation

We evaluate the performance of Ratchet with respect to two aspects: accuracy and usefulness of generated statements. In all of the results presented in this section, the NMT models are trained and tested with data from the same project (i.e., within-project evaluation).

Can the models generate valid statements?

We consider complete and parsable statements as valid statements. We investigate whether the generated statements are valid using the same process of step (5) in the preprocessing described in Section 4.2. Table 5 shows the number (and percentage) of generated valid statements. We do not use thresholds here, that is, all generated statements including low scores are considered. As we see from the table, in most cases the models generated valid and complete Java statements. These high accuracy results are especially interesting since we did not explicitly teach the models the Java language specification. Simply, the models were able to achieve this high level of performance by themselves, using approx. 1,500 to 10,000 statement pairs.

> In most cases of the five projects nearly 100% of the generated statements are valid Java statements. In total the models generated 230 valid statements for 233 queries (98.7%).

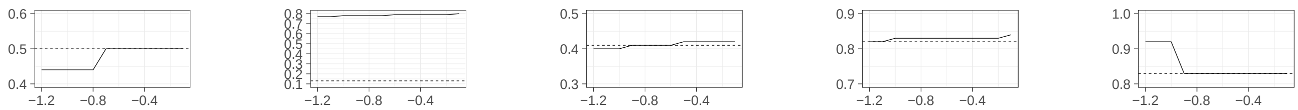
How accurate are the generated statements?

In this section we evaluate the accuracy in a strict manner, that is, only generated statements that are **identical to references** are considered as correct. Our results are based on the NU (no unknown) category of statements, since as mentioned earlier, other categories are difficult (impossible for UR) to generate accurate statements that are identical to the reference statements.

Before analyzing accuracy, we categorize the outputs into four types:

- **Correct:** a generated statement is identical to the reference, including arguments.
- **Argument incorrect:** a generated statement is identical to the referecne, except for the arguments.
- **Incorrect:** a generated statement is not identical to the reference, even if we exclude the arguments.
- **NA:** a generated statement is invalid or identical to the query, or its score is lower than a threshold.

11. Commit IDs in a historage and the corresponding commit IDs in the original Git repository are different because the contents are different. But we can trace the corresponding original commit IDs from historage since they are written in `git notes` of historage.



(a) ambari (b) camel (c) hadoop (d) jetty (e) wicket

Fig. 4. F_1 values with thresholds (-1.2 to -0.1). The solid lines are F_1 values of Ratchet and the dotted lines are F_1 values of the baseline.

To measure the accuracy of generated results, we compute precision, recall, and F_1, which are defined as: $precision = \frac{\#correct}{\#provided}$, $recall = \frac{\%correct}{\#queries}$, and $F_1 = \frac{2 \times precision \times recall}{precision + recall}$, where $\#provided$ is the sum of $\#correct$, $\#argument\ incorrect$, and $\#incorrect$. Higher precision indicates that the provided results are correct. Higher recall means that the results contain less NA but many correct. Providing a small number of results with high confidence can improve precision but lower recall. Since there is a tradeoff between precision and recall, F_1, the harmonic mean of precision and recall, is also presented.

We compare the accuracy of our NMT models with the pattern-based patch suggestion approach, i.e., patch suggestion with line-granular snippets that already exist in the training data, which serves as our baseline [51]. To do so, we examine whether a query statement exists in the pre-correction statement training data corpus, and if it does, we check whether the corresponding post-correction statement also exists (in the training data). If this happens, then we consider the statement to be covered by the Plastic Surgery approach. If there is no identical pre-correction statement, we mark the result as NA. Note that this baseline is slightly different from the Plastic Surgery hypothesis since it searches post-correction statements from all packages, which requires the existence of pre-correction statements in the codebase. We apply the same argument replacing processing in Section 4.3 to make it fair.

When evaluating Ratchet and as stated in Section 4.3, we use a threshold to ignore results with low confidence. Figure 4 illustrates the transitions of F_1 values with different thresholds (from -1.2 to -0.1). The solid lines are F_1 values of Ratchet and the dotted lines are F_1 values of the baseline, which do not change with thresholds. We find that F_1 values slightly change when we vary the thresholds. Lowering thresholds improves recall, however, it impacts the precision in the opposite direction. On the other hand, raising the threshold improves precision but makes recall worse.[12] Based on our analysis of the threshold, we empirically set the threshold as -0.7 for the analyses that follow.

Table 6 shows the results of our approach and compares it with the results of the baseline. We observe that, as reported in the Plastic Surgery hypothesis paper [51], the baseline of the pattern-based patch recommendation works in many cases, that is, changes (corrections) contain snippets that already exist in code repositories at the time of the changes, and these snippets can be efficiently found and exploited. That said, Table 6 shows that Ratchet improves the results in two projects and does not change in three projects. We observe that in camel, the results are greatly improved: 26 correct statements are generated compared with one correct recommendation from the pattern matching of the baseline. Our results show that the NMT models work, as well as the baseline, if there are easily exploited statement-level patterns (i.e., reusable snippets), and works better than the baseline if there exist only finer-grained exploited patterns (i.e., fine-grained fixing patterns), which the statement-based pattern matching cannot use.

Table 7 presents examples of generated fixes that cannot be fixed by the baseline, but have a fix generated with our models. Sometimes the model learns the incrementation of value (query 1). Generics-related fixes are typical examples

12. The data of wicket is an exception, in which we can find a correct result without increasing incorrect statements when lowering the threshold.

TABLE 6
Fix generation for *buggy* queries. Threshold is -0.7. Bold indicates win in comparison with the baseline.

		Ratchet			Baseline		
amb.	correct	2		(33%)	2		(33%)
	arg incor.	0		(0%)	0		(0%)
	incor.	0		(0%)	0		(0%)
	NA	4		(67%)	4		(67%)
	Pr, Re, F	1.00	0.33	0.50	1.00	0.33	0.50
cam.	correct	26		(62%)	1		(2%)
	arg incor.	2		(5%)	2		(5%)
	incor.	2		(5%)	1		(2%)
	NA	12		(29%)	38		(90%)
	Pr, Re, F	**0.87**	**0.62**	**0.72**	0.25	0.02	0.04
had.	correct	7		(24%)	7		(24%)
	arg incor.	3		(10%)	3		(10%)
	incor.	10		(34%)	10		(34%)
	NA	9		(31%)	9		(31%)
	Pr, Re, F	0.35	0.24	0.29	0.35	0.24	0.29
jet.	correct	101		(68%)	101		(68%)
	arg incor.	11		(7%)	11		(7%)
	incor.	9		(6%)	11		(7%)
	NA	28		(19%)	26		(17%)
	Pr, Re, F	**0.83**	0.68	**0.75**	0.82	0.68	0.74
wic.	correct	5		(71%)	5		(71%)
	arg incor.	0		(0%)	0		(0%)
	incor.	0		(0%)	0		(0%)
	NA	2		(29%)	2		(29%)
	Pr, Re, F	1.00	0.71	0.83	1.00	0.71	0.83

TABLE 7
Examples of generated statements that cannot be fixed by the baseline.

	Query statement	Generated statement
1.	commands [10] = this . passwordFile . toString ( ) ;	commands [11] = this . passwordFile . toString ( ) ;
2.	List body = assertIsInstanceOf (arg†) ;	List <? > body = assertIsInstanceOf (arg†) ;
3.	Set < String > knownRoles = new HashSet ( ) ;	Set < String > knownRoles = new HashSet <> ( ) ;
4.	return this . height ;	return height ;

arg†: List . class , result . getExchanges ( ) . get (0) . getIn ( ) . getBody ( )

of successful generation with the NMT models (query 2 and 3). Sometimes it is preferred to remove `this` (query 4) if it makes the style consistent with the styling used in the specific project. In fact, our models learned to remove the keyword 'this' because similar patterns were prevalent in the project's history.

> NMT-based patch generation works better than pattern-based patch suggestion, achieving F_1 values between 0.29 to 0.83 for buggy queries. In total 157 correctly generated statements without method arguments, the contents of method arguments for 141 statements (89.8%) are correctly provided by reusing the contents of method arguments in queries.

Do humans detect similarity between generated statements and actual statements?

During the previous evaluation of accuracy, we considered the generated statements to be correct only if they are *identical to the reference statements*, otherwise they are considered to be incorrect or NA. To investigate whether the generated statements are useful, even if they are not identical to actual future corrections, we also perform a human evaluation with such (non-identical) corrections.

We show survey participants the following three code snippets for one fix: i) an original problematic code snippet (before correction), ii) the actually fixed code snippet (after correction), and iii) a code snippet that is proposed as a fix by our NMT models. All code snippets contain one type of buggy or fixed statements with the surrounding statements.[13]

From the five projects, we collect ten corrections including five correctly and five incorrectly generated statements in the NU (no unknown) category, which are evaluated in Table 6. In addition, we collect five fixes that belong to the UQ (unknown in query) or the UR (unknown in reference) categories, which are known to be difficult for NMT models to generate. For simplicity, we call the above three groups *correct fixes*, *incorrect fixes*, and *challenging fixes* respectively.

For each correction we prepare the following four statements, and ask the participants to evaluate using a five-level Likert scale scores from 1 (strongly disagree) to 5 (strongly agree) whether: (a) *The proposed fix helps you to understand the required change*, (b) *The proposed fix can be a reference if you were to create your own fix*, (c) *The proposed fix is harmful or confusing*, and (d) *The proposed fix does not make sense and I will just ignore it*. We asked not only positive impressions but also negative impressions in order to assess the usefulness and potential risks of incorrect generation. The survey material is available online.[14]

We recruited participants in Canada, US, and Japan, and 20 people participated in the survey including five undergraduate, 14 graduate students, and one professor. As Siegmund *et al.* reported that self estimation seems to be a reliable way to measure programming experience [86], we asked the participants to estimate their experience in both, overall and Java programming experience. The participants can select any of 5 choices, varying between 1 (very inexperienced) to 5 (very experienced). Those who score 4 or 5 in both self estimation are considered to be *high*-experienced and others are considered to have *low*-experience. Five in six high experienced participants have more than five years of development experience, and the other have three-to-five years of experience. In 14 low-experience participants, the experience periods vary from less than one year, one-to-three years, three-to-five years, and more than five years.

Figure 5 shows the result of the *correct fix* group. The results shown in the figure show that the generated statements are useful. All high-experience and most of the low-experience participants agreed (scores 4 or 5 for questions (a) and (b)) that the correct fix statements helped them and that the statements and did not have negative effects (i.e., most scores are 1 or 2 for questions (c) and (d)).

Figure 6 shows the results of the *incorrect fix* group, which includes statements with incorrect method calls and/or incorrect generic types, for example. We assumed that these fixes are harmful or confusing because they tend to be partially the same as the references, but are slightly different from the actual fixes. However, as evidenced by the results shown in Figure 6, the majority of highly and low experienced participants agree to that such imperfect statements may still be helpful (i.e., by providing positive answers to statement (a) and (b)). Although the highly experienced participants tend to consider such imperfect statements harmful or confusing (highly experienced agreed 40% and low experienced agreed 24%), both high and low experienced participants did not consider the proposed fix did not make sense.

The following are some comments we received: "*A potentially better fix than original*," "*I prefer having a 'this' but this is personal preference.*," "*The word 'info' seems more clear*

13. One case has two buggy or fixed statements that are similar to each other, and others only have one statement of buggy or fixed statement.

14. Survey material for human evaluation: https://tinyurl.com/RachetSurvey

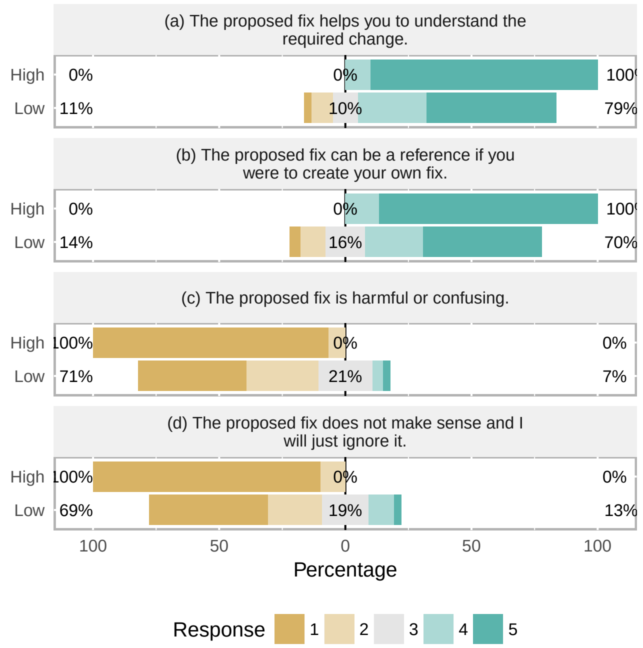


Fig. 5. Survey results of five *correct fixes*. Responses are Likert scale from 1 (strongly disagree) to 5 (strongly agree).

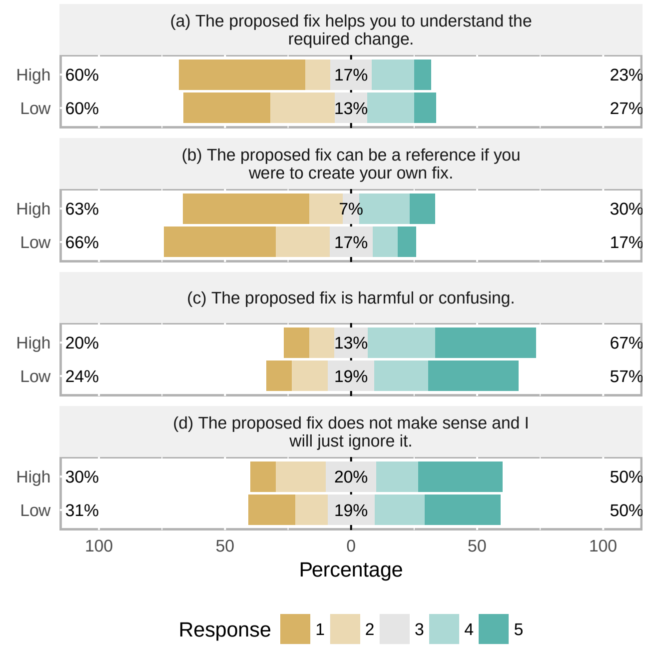


Fig. 7. Survey result of five *challenging fixes* with six high and 14 low experienced participants.

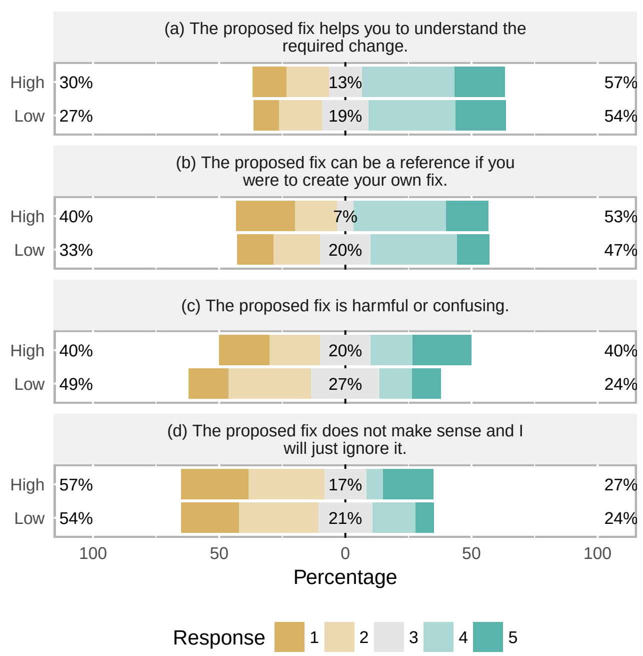


Fig. 6. Survey result of five *incorrect fixes* with six high and 14 low experienced participants.

than 'trace.' Good change," and "*Changed to a wrong direction*. One participant pointed out that the proposed fixes seem to provide several pieces of information, for example, the location of fix, the need of initialization of methods, and types for generics. S/he claimed that this information is useful if s/he knows the context of the code, even if an error exists. We find that even for the same fixes, some participants perceived them differently, from which we can infer that sometimes better fixes depend on preferences and/or the context.

Figure 7 shows the result of the *challenging fix* group. The fixes belonging to this group are difficult to generate because of unknown terms, which means that the queries and correct answers are mostly unseen by the models. Therefore we considered that those fixes did not make sense and did not provide any useful information. One case is changing `BigInteger` to `toHexString`, which even fails the compilation check. A participant left a comment "*I think it would produce more confusion than help.*" As seen from the figure, the majority of both, highly and low experienced participants have negative impressions with regards to such statements. However, some still have positive opinions even for such failure cases. Another case is given a query of '`FSDataOutputStream fos = null ;`,' generating '`HdfsDataOutputStream copyError = null ;`' while the correct answer is '`HdfsDataOutputStream fos = null ;`.' This happens because the term '`fos`' does not appear in the target (i.e., post-correction statement) corpus. In the training corpora, there is no statement co-occurring `FSDataOutputStream` with `fos` or `null`. The model learned the replacement of `FSDataOutputStream` and `HdfsDataOutputStream` from the different context of statements. For this case, the participants evaluated more positively than negatively, although there were some comments which stated that the generated statements can be confusing. In sum, we find that even for the challenging fixes, they might be useful.

TABLE 8
Summary of testing data from not bug-fixing statements.

	ambari	camel	hadoop	jettty	wicket
test year	2014	2014	2013	2014	2015
NU	64	114	135	151	28
UQ	792	386	1,274	428	99
UR	38	46	113	148	7

TABLE 9
Number of the generated valid statements for *non-buggy* queries.

ambari	camel	hadoop	jetty	wicket
63/64 (98%)	113/114 (99%)	134/135 (99%)	149/151 (99%)	20/28 (71%)

Even if generated fixes are not identical to actual fixes, they can be helpful because they can suggest the locations of required changes and possible replacements/insertions/deletions. Sometimes better fixes depend on personal preferences or the styles of projects. Although NMT models can learn fine-grained patterns of changes, the lack of information or novel queries are major challenges of fix generation.

7 Discussions

7.1 Generating Non-Bug-Fixing Statements

For the accuracy evaluation in Section 6, we only considered bug-fixing statements. Here we investigate the applicability of Ratchet in a more general context, i.e., for non-bug-fixing statements as well. In the same test year, we collected non-bug-fixing statements as shown in Table 8. Again, we use a similar setup as we did for bug-fixing statement evaluations and compare the generated statements with the baseline.

Table 9 shows the number of generated valid statements. Similar to the result for buggy queries in Table 5, most generated statements are valid Java statements. Table 10 shows the results for non-bug-fixing statements. The F_1 values for non-bug-fixing queries ranges between 0.07 to 0.49. These F_1 values are lower than the results obtained for the bug-fixing queries shown in Table 6. That said, we still observe that in all five projects, Ratchet outperforms the baseline. One possible explanation for the lower performance is the fact that there are relatively more UQ and UR statement pairs for non-bug-fixing datasets (as seen by comparing Table 4 and Table 8), which indicates the unique nature of non-bug-fix changes.

7.2 Applying to Bug Dataset

We also investigate the applicability of Ratchet on a commonly used bug dataset, Defects4J [87].[15] The same steps

15. We used the version 2.0.0 from https://github.com/rjust/defects4j.

TABLE 10
Fix generation for *non-buggy* queries. Threshold is -0.7. Bold indicates win in comparison with the baseline.

		Ratchet			Baseline		
amb.	correct	6		(9%)	5		(8%)
	arg incor.	7		(11%)	6		(9%)
	incor.	10		(16%)	17		(27%)
	NA	41		(64%)	36		(56%)
	Pr, Re, F	**0.26**	**0.09**	**0.14**	0.18	0.08	0.11
cam.	correct	33		(29%)	30		(26%)
	arg incor.	3		(3%)	4		(4%)
	incor.	34		(30%)	25		(22%)
	NA	44		(39%)	55		(48%)
	Pr, Re, F	0.47	**0.29**	**0.36**	**0.51**	0.26	0.35
had.	correct	7		(5%)	7		(5%)
	arg incor.	21		(16%)	22		(16%)
	incor.	31		(23%)	28		(21%)
	NA	76		(56%)	78		(58%)
	Pr, Re, F	0.12	0.05	0.07	0.12	0.05	0.07
jet.	correct	15		(10%)	14		(9%)
	arg incor.	4		(3%)	3		(2%)
	incor.	32		(21%)	24		(16%)
	NA	100		(66%)	110		(73%)
	Pr, Re, F	0.29	**0.10**	0.15	**0.34**	0.09	0.15
wic.	correct	10		(36%)	0		(0%)
	arg incor.	0		(0%)	0		(0%)
	incor.	3		(11%)	3		(11%)
	NA	15		(54%)	25		(89%)
	Pr, Re, F	**0.77**	**0.36**	**0.49**	–	–	–

TABLE 11
Characteristics of fixes for the targeted bugs and summary of testing data from Defects4J.

	closure	lang	math	time	mockito
# bugs	151	53	83	20	23
one line	34 (23%)	7 (13%)	30 (36%)	3 (15%)	5 (22%)
only add	40 (26%)	26 (49%)	18 (22%)	7 (35%)	13 (57%)
NU	21	3	19	5	0
UQ	8	2	13	1	0
UR	54	4	17	6	5

shown in Section 4.1, are applied (we only targeted Git repositories), and bug-fixing modifications within methods are identified.

Table 11 presents the characteristics of bugs in the dataset. We see that only 13% to 36% of bugs include single statement (one-line) changes for fixing, which are applicable for Ratchet. From the other bugs, 22% to 57% bugs require only addition of statements, which is not targeted by our current approach. From the bug-fixing commits, using the same steps detailed in Section 4.2, we collected pre- and

TABLE 12
Correct fix generations for the bugs in Defects4J.

bug_id	required fix	result
closure-30	1 one-line and 1 large change	1 statement with incorrect arguments
closure-46	1 one-line	complete fix including arguments
closure-134	4 add, 1 del., 4 one-line, and 3 large changes	1 statement with incorrect arguments

post-correction statements for testing data. Training data was prepared from the other commits. As seen in Table 11, there are 48 statements in the NU category. Similar to other settings, we use the buggy statements as queries.

Ratchet generated one complete fix and two partial fixes, which are shown in Table 12. The bug of closure-46 requires one single statement fix and Ratchet could successfully generate the fix including an argument. The other two bugs require multiple and larger changes, and Ratchet generated *argument incorrect* fixes of single statements. Since the proposed approach does not account for the context of changes, it cannot handle appropriate argument changes. Compared to 26 successful fixes by ELIXIR, one of the recent program repair techniques [21], the performance of Ratchet is low on the Defect4J dataset.

This result highlights the limitations of Ratchet in fixing specific bugs, which require multiple modifications, requires insertions of statements, are not repetitive in code repositories, and so on. CODIT, a tree2tree model-based change generation technique, reported several correct modifications related to method arguments, for Defects4J [88], but it did not generate fixes in Table 12. Hence, we believe that our seq2seq approach can be a complement to tree2tree approaches.

7.3 When/Why Does NMT Fail?

We see that NMT can work better than the pattern-based patch suggestion for learning past changes and generating fixes. However, we also find limitations of our approach using NMT for code repository data. We examined some of the cases where our approach failed and discuss the challenges (and possible improvements) from two aspects, modeling and training.

Modeling: Although NMT can learn the semantic and structural information by taking global context into consideration [89], some limitations of NMT are known and studied [89], [90].

Out-of-vocabulary problem or *UNK problem*. NMT usually uses the top-N most frequent words in the training data and regards other words as unseen ones, *UNK*. Therefore NMT often makes mistakes in translating low-frequency words. While we alleviated this problem somewhat by incorporating dictionaries into our method [91], we still find similar issues in low-frequency or novel identifier names, as discussed in the survey result of *challenging fix*. Since those names can be identified from the context, integrating NMT with program analysis could be a promising direction.

Coverage problem. NMT lacks a mechanism to guarantee that all words in a source statement are translated, and usually favors short translations. For example, in translation of long method chains with low-frequency tokens, we see insufficient outputs including incomplete statements and disappearing method calls. Moreover, this problem makes it difficult to address larger fix generation for more than one line. As there are several studies trying to address this problem [92], [93], [94], we can consider applying these rapidly developed techniques.

Training: In addition to techniques related to NMT, we think there is room for improvement in preparing training data. In this study, we design the experiment as batch learning, that is, whole training data is prepared from the past until the previous year of the test year. However, Barr *et al.* reported that more reusable pieces of code exist in the immediately previous version [51]. Previous studies have tried an online learning setting called training on errors [95], [96]. Applying such online learning could be a promising challenge too.

7.4 Limitations and Challenges

Limitations of Ratchet are summarized as follows.

1) It cannot generate patches that need additions or deletions of statements.
2) It cannot generate patches that consist of multiple statements.
3) It cannot generate patches that include the changes of method or array arguments.
4) It cannot account for the context of patches outside statements.
5) It cannot generate patches that include unknown tokens.
6) It cannot generate assembled patches for single bugs.
7) It is not evaluated in a cross-project setting.
8) It does not include a bug-localization step.

Considering the amount of hunks with single-statement pairs in Table 2 (66.3% in average), the filtering results in Table 3 (40.4% in average), the percentage of NU in Table 4 (41.0% in average), around 11% of change hunks can be targeted in our approach.

Regarding 1), 2), 3), and 4), extending the granularity of code to be learned can address these issues to some extent. As we discuss in Section 8.1, some studies reported that learning the contents of methods or classes can work with seq2seq models [97], [98]. More advanced tree-to-tree models may also applicable to address these limitations [88]. Regarding 5), increasing appropriate training data is one direction, as well as accounting for context information. Regarding 6), recent studies have examined multi-hunk program repair [99]. Our fine-grained code history analysis could be suitable for their approach of using revision histories. Although this paper does not conduct the evaluation of cross-project learning (limitation 7)), Ratchet can learn from data from multiple projects. Selecting and preprocessing data from multiple projects in an appropriate way can be a practical challenge.

Regarding 8), there are studies of fine-grained defect prediction [100], [101], [102]. To localize problematic statements, we could try applying these techniques. We could also consider training another neural network to identify problematic statements to be query statements to Ratchet.

7.5 Threats to Validity

Concerning *external validity*, this study only targets five open source projects written in Java. As we do not have clear selection criteria, there can be a selection bias. Projects with different sizes, different management governance, etc. can lead to different results. Regarding programming languages, there is a threat of generalization, and it should be interesting to extend this study to other languages.

With respect to *construct validity*, we collect fixes from histories, which can contain mistakes. For example, sometimes fixes can be reverted, but we do not consider such intention. In addition, the SZZ algorithm used for identifying bug-inducing and bug-fixing commits is known to produce errors [103]. Although we do not distinguish buggy and non-buggy changes for training, we classify test data as buggy or non-buggy. This could impact our discussion regarding the type of changes. However, as presented in Section 7.1, Ratchet can work for generation of non-bug-fixing statements as well as bug-fixing statements.

Another threat to *construct validity* exists in our preprocessing. We removed short statements (step (3)), the contents of method and array arguments (step (4)), unparsable statements (step (5)), and older and less frequent post-correction statements from multiple candidates (step (6)), to ignore noises. Although these steps were prepared in our trial-and-error experiments and evaluated empirically, different parameters and processes may improve the performance. Exploring a better configuration of preprocessing can be practical future work.

To mitigate threats to *reliability*, we made our dataset and survey material publicly available (see Section 5.3 and Section 6).

8 Related Work

In this section, we discuss similar NMT-based patch generation studies, and then discuss two research areas, namely probabilistic models of source code and change mining, which are typically used to build models by learning and mining data for learning.

8.1 NMT-based Patch Generation

Learning source code changes using NMT-based techniques, NMT-based automated code changes, is an emerging research topic. Tufano *et al.* conducted empirical studies to investigate the feasibility of learning bug-fixing patches using NMT techniques [104], [105]. Similar to our approach, they built seq2seq models. By extending those studies, Tufano *et al.* studied the ability of a seq2seq NMT model to automate code changes for pull requests [97]. One of the differences of the earlier approaches and Ratchet is the granularity of code to be learned. While the above studies targeted methods within 50 or 100 tokens, Ratchet targets statements. Positive results at both granularity leveles show the capability of NMT models to learn different types of code changes. Although Tufano *et al.* prepared their training and testing data by random partitioning [97], we prepared data considering chronological order, to emulate practical scenarios. Tufano *et al.* largely abstracted tokens for cross-project learning, while Ratchet kept identifiers and literals except for arguments to learn project-specific time-sensitive changes, which results in the successful number incrementation shown at the first result in Table 7.

SequenceR is another NMT-based system to learn source code changes based on a seq2seq model and copy mechanism [98]. Similar to Ratchet, one-line changes are targeted for fixing. While Ratchet expects only a buggy line as an input, SequenceR accepts surrounding method and class as well as an annotated buggy line as the context. SequenceR is an end-to-end approach including validation with test cases. CODIT learns source code change patterns with tree-to-tree NMT models considering AST-level changes [88]. Similar to Tufano *et al.* [97], cross-project datasets without considering time were used for the evaluation of both SequenceR and CODIT.

While making the vocabulary small is considered to be one of challenges in other studies [88], [97], [98], we did not explicitly limit the number of tokens or identifiers to be learned. There seems to be a trade-off relationship between the vocabulary size and the context size. Since we targeted almost the smallest context (single-statement changes and changes within a single project), we did not need to make the vocabulary small. Addressing this trade-off to consider both larger vocabulary and wider context will be challenging future work. Another major difference between this study and the other studies is our human evaluation with the survey. We observed that sometimes the survey participants perceived positively even if generated statements were not identical to actual statements. Further user studies in practical scenario could be another future challenge.

8.2 Probabilistic Models of Source Code

There are several studies on probabilistic machine learning models of source code for different applications using different techniques. Allamanis *et al.* conducted a large survey on this topic [106]. Table 13 is originally presented in the survey of representative code models [106]. From the original table, non-refereed papers are excluded, some missing papers are added, and the column Data is newly prepared, which summarizes analyzed data in terms of programing languages, data sources, and historical information.

As we see from the table, probabilistic machine learning models have been studied for various applications, such as code completion, code synthesis, coding conventions, and so on. From the point of view of models, newer techniques of neural networks (NN), especially neural seq2seq models, have not been extensively studied yet. So there are possibilities of extending and improving previous studies applying these models.

From the data column, we see that several programing languages have been studied including Java, C, C#, JavaScript, Python, among others. Although most of studies collected data from code repositories, some used other data

TABLE 13
Studies on source code generating models. The column *Data* is presented by the authors and other contents were previously presented by Allamanis *et al.* [106]. Only referred papers are presented.

Study	Representation	Model	Application	Data
Allamanis and Sutton [107]	Token	n-gram	—	Snapshot (Java)
Allamanis *et al.* [108]	Token + Location	n-gram	Coding conventions	Snapshot (Java)
Allamanis and Sutton [109]	Syntax	Grammar (pTSG)	—	Snapshot (Java)
Allamanis *et al.* [110]	Syntax	Grammar (NN-LBL)	Code search/synthesis	Stack Overflow (C# and NL)
Bielik *et al.* [111]	Syntax	PCFG + annotations	Code completion	Snapshot (JavaScript)
Campbell *et al.* [112]	Token	n-gram	Syntax error detection	Selected versions (Java)
Cerulo *et al.* [113]	Token	Graphical model (HMM)	Information extraction	Snapshot (Java) and NL
Cummins *et al.* [114]	Character	NN (LSTM)	Benchmark synthesis	Benchmarks (OpenCL)
Gulwani and Marron [115]	Syntax	Phrase model	Text-to-code	Created (DSL and NL)
Gvero and Kuncak [116]	Syntax	PCFG + Search	Code synthesis	Created (Java and NL)
Hata *et al.* [96]	Token	Orthogonal sparse bigrams	Bug prediction	Long period (Java)
Hata *et al.* [117]	Token	Vector space model	Bug prediction	Snapshot (Java)
Hellendoorn *et al.* [118]	Token	n-gram	Code review	Short period (Java)
Hellendoorn and Devanbu [119]	Token	n-gram (cache)	—	Snapshot (Java)
Hindle *et al.* [120]	Token	n-gram	Code completion	Snapshot (Java and C)
Hsiao *et al.* [121]	PDG	n-gram	Program analysis	Snapshot (JavaScript)
Ling *et al.* [42]	Token	RNN + attention	Code synthesis	Snapshot (Java and Python)
Karaivanov *et al.* [122]	Token	Phrase	Migration	Snapshot (C# and Java)
Kushman and Barzilay [123]	Token	Grammar (CCG)	Code synthesis	Created (Regex and NL)
Maddison and Tarlow [124]	Syntax with scope	NN	—	TopCoder (C#)
Menon *et al.* [125]	Syntax	PCFG + annotations	Code synthesis	Excel help forums
Mizuno and Kikuno [95]	Token	Orthogonal sparse bigrams	Bug prediction	Long period (Java)
Nguyen *et al.* [126]	Token + parse info	n-gram	Code completion	Snapshot (Java)
Nguyen *et al.* [127]	Token + parse info	Phrase SMT	Migration	Snapshot (Java and C#)
Nguyen and Nguyen [128]	Partial PDG	n-gram	Code completion	Snapshot (Java and C#)
Nguyen *et al.* [129]	Bytecode	Graphical model (HMM)	Code completion	Android
Oda *et al.* [50]	Syntax + token	Tree-to-string + phrase	Pseudocode generation	Created (Python and NL)
Rabinovich *et al.* [44]	Syntax	NN (LSTM-based)	Code synthesis	Snapshot (Java and Python)
Ray *et al.* [102]	Token	n-gram (cache)	Bug detection	Short period (Java)
Raychev *et al.* [130]	Token + constraints	n-gram / RNN	Code completion	Android
Raychev *et al.* [131]	Syntax	PCFG + annotations	Code completion	Snapshot (JavaScript)
Sharma *et al.* [132]	Token	n-gram	Information extraction	Stack Overflow and Twitter
Tu *et al.* [133]	Token	n-gram (cache)	Code completion	Snapshot (Java and Python)
Vasilescu *et al.* [134]	Token	Phrase SMT	Deobfuscation	Snapshot (JavaScript)
White *et al.* [135]	Token	NN (RNN)	—	Snapshot (Java)
Yadid and Yahav [136]	Token	n-gram	Information extraction	Android tutorial videos
Yin and Neubig [43]	Syntax	NN (seq2seq)	Synthesis	Snapshot (Python and DSL)
Ratchet	Syntax	NN (seq2seq)	Patch generation	Long period (Java)

sources, for example, programs in TopCorder.com [124], Microsoft Excel help forums [125], Android programming tutorial videos [136], to build probabilistic models of source code. From source code repositories, collecting source code in selected snapshots is a common procedure. However, when considering software evolution, that is, software is updated continuously, learning over long periods is more practical. As discussed in Section 7.3, online machine learning is one of challenges in this scenario. Previous studies demonstrated learning methods in long periods, called training on errors [95], [96]. This can be a good hint for future research on online machine learning of patch generation.

8.3 Change Mining

Analyzing and exploiting historical change patterns is another similar topic to this work. Kim *et al.* proposed bug finding techniques based on textual code change histories [137]. From the analysis of open source repositories, they reported that a large amount of bugs appeared repeatedly. From the analysis of graph-based object usage models, Nguyen *et al.* also reported recurring bug-fix patterns and demonstrated fix recommendation based on those patterns [138]. To make use of similar code changes, Meng *et al.* proposes a tool called LASE for creating and applying context-aware edit scripts [139]. LASE analyzes AST-level changes and generates AST node edit operations. From a large-scale study of AST-level code changes in multiple

Java projects, Nguyen *et al.* reported that repetitiveness is high for small size changes and similar bug-fix changes repeatedly occurred in cross projects [80]. Barr *et al.* studied the Plastic Surgery hypothesis, that is, same changes already exist in code histories and those changes can be efficiently found and exploited [51]. From line-granular snippet matching analyses, they reported that changes are repetitive and this repetitiveness is usefully exploitable. Yue *et al.* reported, from an empirical study of large-scale bug fixes, that 15-20% of bugs involved repeated fixes [140].

As these studies presented, using change patterns can be promising. However, from the study of the uniqueness of changes, instead of common changes, Ray *et al.* reported that unique changes are more common than non-unique changes [56]. This implies that simply applying past change patterns has limited capabilities in terms of reuse. As our results demonstrated, NMT-based learning approaches have the ability to address this issue by learning bug-fix correspondences on a variety of levels.

9 Conclusion

In this paper, we introduced Ratchet, an NMT-based technique to generate bug fixes from past fixes. Through an empirical validation on five open source projects, we find that Ratchet is effective in generating fixes. Moreover, we show that Ratchet can even be used to generate statements for non-bug-fixing statements. We compare Ratchet to pattern-based patch suggestion as a baseline and show that Ratchet performs at least as well as the baseline.

We also investigate cases where Ratchet fails and find that Ratchet, or more generally NMT, suffers from the out-of-vocabulary problem since it depends on the presence of words in the past to train on. Also, NMT cannot guarantee that all words are covered/translated. These aforementioned issues are areas that we plan to tackle in future work.

Acknowledgments

References

[1] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," *IEEE Trans. Softw. Eng.*, vol. 40, no. 5, pp. 427–449, May 2014. [Online]. Available: https://doi.org/10.1109/TSE.2014.2312918

[2] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1109/TSE.2011.104

[3] 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 *Proc. of 34th Int. Conf. on Softw. Eng.*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 3–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337225

[4] P. Liu and C. Zhang, "Axis: Automatically fixing atomicity violations through solving control constraints," in *Proc. of 34th Int. Conf. on Softw. Eng.*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 299–309. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337259

[5] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proc. of 35th Int. Conf. on Softw. Eng.*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 772–781. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486890

[6] Z. Coker and M. Hafiz, "Program transformations to fix c integers," in *Proc. of 35th Int. Conf. on Softw. Eng.*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 792–801. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486892

[7] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Proc. of 28th IEEE/ACM Int. Conf. on Automated Softw. Eng.*, ser. ASE'13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 356–366. [Online]. Available: https://doi.org/10.1109/ASE.2013.6693094

[8] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proc. of 36th Int. Conf. on Softw. Eng.*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 254–265. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568254

[9] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, "Minthint: Automated synthesis of repair hints," in *Proc. of 36th Int. Conf. on Softw. Eng.*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 266–276. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568258

[10] F. S. Ocariza, Jr., K. Pattabiraman, and A. Mesbah, "Vejovis: Suggesting fixes for javascript faults," in *Proc. of 36th Int. Conf. on Softw. Eng.*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 837–847. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568257

[11] P. Liu, O. Tripp, and C. Zhang, "Grail: Context-aware fixing of concurrency bugs," in *Proc. of 22nd ACM SIGSOFT Int. Symp. on Found. of Softw. Eng.*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 318–329. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635881

[12] Y. Lin and S. S. Kulkarni, "Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability," in *Proc. of 23rd Int. Symp. on Softw. Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 237–247. [Online]. Available: http://doi.acm.org/10.1145/2610384.2610398

[13] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *Proc. of 37th Int. Conf. on Softw. Eng. - Vol. 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 448–458. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818811

[14] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei, "Safe memory-leak fixing for c programs," in *Proc. of 37th Int. Conf. on Softw. Eng. - Vol. 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 459–470. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818812

[15] S. H. Tan and A. Roychoudhury, "Relifix: Automated repair of software regressions," in *Proc. of 37th Int. Conf. on Softw. Eng. - Vol. 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 471–482. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818813

[16] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proc. of 10th Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 166–178. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786811

[17] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proc. of 38th Int. Conf. on Softw. Eng.*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 691–701. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884807

[18] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 34–55, Jan. 2017. [Online]. Available: https://doi.org/10.1109/TSE.2016.2560811

[19] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proc. of 39th Int. Conf. on Softw. Eng.*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 416–426. [Online]. Available: https://doi.org/10.1109/ICSE.2017.45

[20] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: Syntax- and semantic-guided repair synthesis via programming by examples," in *Proc. of 11th Joint Meeting on Found. of Softw. Eng.*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 593–604. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106309

[21] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object oriented program repair," in *Proc. of 32nd IEEE/ACM Int. Conf. on Automated Softw. Eng.*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 648–659. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155643

[22] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *Proc. of 40th Int. Conf. on Softw. Eng.*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 12–23. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180245

[23] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation," in *Proc. of 40th Int. Conf. on Softw. Eng.*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 129–139. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180247

[24] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 17:1–17:24, Jan. 2018. [Online]. Available: http://doi.acm.org/10.1145/3105906

[25] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Trans. Softw. Eng.*, pp. 1–1, 2018.

[26] Y. Jia, K. Mao, and M. Harman. (2018) Finding and fixing software bugs automatically with sapfix and sapienz. [Online]. Available: https://code.fb.com/developer-tools/finding-and-fixing-software-bugs-automatically-with-sapfix-and-sapienz/

[27] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proc. of 10th Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 532–543. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786825

[28] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *Proc. of 38th Int. Conf. on Softw. Eng.*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 702–713. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884872

[29] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proc. of 40th Int. Conf. on Softw. Eng.*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180233

[30] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proc. of 35th Int. Conf. on Softw. Eng.*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 802–811. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486893

[31] M. Martinez and M. Monperrus, "Mining software repair models for reasoning on the search space of automated program fixing," *Empirical Softw. Eng.*, vol. 20, no. 1, pp. 176–205, Feb. 2015. [Online]. Available: http://dx.doi.org/10.1007/s10664-013-9282-8

[32] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *Proc. of 30th IEEE/ACM Int. Conf. on Automated Softw. Eng.*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 295–306. [Online]. Available: https://doi.org/10.1109/ASE.2015.60

[33] X. B. D. Le, D. Lo, and C. L. Goues, "History driven program repair," in *Proc. of 23rd IEEE Int. Conf. on Softw. Analysis, Evolution and Reengineering*, vol. 1, March 2016, pp. 213–224.

[34] E. C. Campos and M. A. Maia, "Common bug-fix patterns: A large-scale observational study," in *Proc. of 11th ACM/IEEE Int. Symp. on Empirical Softw. Eng. and Measurement*, ser. ESEM '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 404–413. [Online]. Available: https://doi.org/10.1109/ESEM.2017.55

[35] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proc. of 11th Joint Meeting on Found. of Softw. Eng.*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 727–739. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106253

[36] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *Proc. of 32nd IEEE/ACM Int. Conf. on Automated Softw. Eng.*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 660–670. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155644

[37] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proc. of 27th ACM SIGSOFT Int. Symp. on Softw. Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: ACM, 2018, pp. 298–309. [Online]. Available: http://doi.acm.org/10.1145/3213846.3213871

[38] W. Chan, N. Jaitly, Q. Le, and O. Vinyals, "Listen, attend and spell: A neural network for large vocabulary conversational speech recognition," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. IEEE, 2016, pp. 4960–4964.

[39] O. Vinyals, Ł. Kaiser, T. Koo, S. Petrov, I. Sutskever, and G. Hinton, "Grammar as a foreign language," in *Proceedings of the 29th Annual Conference on Neural Information Processing Systems (NIPS)*, 2015, pp. 2773–2781.

[40] A. M. Rush, S. Chopra, and J. Weston, "A neural attention model for abstractive sentence summarization," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2015, pp. 379–389. [Online]. Available: http://aclweb.org/anthology/D15-1044

[41] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2016, pp. 2073–2083.

[42] W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. Kočiský, F. Wang, and A. Senior, "Latent predictor networks for code generation," in *Proc. of 54th Annual Meeting of Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, August 2016, pp. 599–609. [Online]. Available: http://www.aclweb.org/anthology/P16-1057

[43] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proc. of 55th Annual Meeting of Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, July 2017, pp. 440–450. [Online]. Available: http://aclweb.org/anthology/P17-1041

[44] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," in *Proc. of 55th Annual Meeting of Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, July 2017, pp. 1139–1149. [Online]. Available: http://aclweb.org/anthology/P17-1105

[45] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proc. of 24th ACM SIGSOFT Int. Symp. on Found. of Softw. Eng.*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 631–642. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950334

[46] S. Bhatia and R. Singh, "Automated correction for syntax errors in programming assignments using recurrent neural networks," in *Proc. of 2nd Indian Workshop on Machine Learning*, 2016.

[47] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *AAAI Conference on Artificial Intelligence*, 2017. [Online]. Available: https://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603

[48] P. Koehn, *Statistical Machine Translation*. Cambridge Press, 2010.

[49] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 651–654.

[50] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation (t)," in *Proc. of 30th IEEE/ACM Int. Conf. on Automated Softw. Eng.*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 574–584. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.36

[51] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proc. of 22nd ACM SIGSOFT Int. Symp. on Found. of Softw. Eng.*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 306–317. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635898

[52] 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 *Proc. of 39th Int. Conf. on Softw. Eng.*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 404–415. [Online]. Available: https://doi.org/10.1109/ICSE.2017.44

[53] R. van Tonder and C. L. Goues, "Static automated program repair for heap properties," in *Proc. of 40th Int. Conf. on Softw. Eng.*, ser.

ICSE '18. New York, NY, USA: ACM, 2018, pp. 151–162. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180250

[54] K. Liu, D. Kim, T. F. Bissyande, S. Yoo, and Y. Le Traon, "Mining fix patterns for findbugs violations," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[55] M. Motwani, S. Sankaranarayanan, R. Just, and Y. Brun, "Do automated program repair techniques repair hard and important bugs?" *Empirical Softw. Eng.*, vol. 23, no. 5, pp. 2901–2947, Oct 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9550-0

[56] B. Ray, M. Nagappan, C. Bird, N. Nagappan, and T. Zimmermann, "The uniqueness of changes: Characteristics and applications," in *Proc. of 12th Work. Conf. on Mining Softw. Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 34–44. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820526

[57] N. Kalchbrenner and P. Blunsom, "Recurrent continuous translation models," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2013, pp. 1700–1709.

[58] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the 28th Annual Conference on Neural Information Processing Systems (NIPS)*, 2014, pp. 3104–3112.

[59] G. Neubig, "Neural machine translation and sequence-to-sequence models: A tutorial," *arXiv preprint arXiv:1703.01619*, 2017.

[60] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[61] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[62] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, no. 3, p. 1, 1988.

[63] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[64] G. Neubig, C. Dyer, Y. Goldberg, A. Matthews, W. Ammar, A. Anastasopoulos, M. Ballesteros, D. Chiang, D. Clothiaux, T. Cohn, K. Duh, M. Faruqui, C. Gan, D. Garrette, Y. Ji, L. Kong, A. Kuncoro, G. Kumar, C. Malaviya, P. Michel, Y. Oda, M. Richardson, N. Saphra, S. Swayamdipta, and P. Yin, "Dynet: The dynamic neural network toolkit," *arXiv preprint arXiv:1701.03980*, 2017.

[65] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[66] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[67] R. Socher, E. H. Huang, J. Pennin, C. D. Manning, and A. Y. Ng, "Dynamic pooling and unfolding recursive autoencoders for paraphrase detection," in *Proceedings of the 25th Annual Conference on Neural Information Processing Systems (NIPS)*, 2011, pp. 801–809.

[68] P.-S. Huang, X. He, J. Gao, L. Deng, A. Acero, and L. Heck, "Learning deep structured semantic models for web search using clickthrough data," in *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*. ACM, 2013, pp. 2333–2338.

[69] T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2015, pp. 1412–1421.

[70] J. Gu, Z. Lu, H. Li, and V. O. Li, "Incorporating copying mechanism in sequence-to-sequence learning," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, August 2016, pp. 1631–1640. [Online]. Available: http://www.aclweb.org/anthology/P16-1154

[71] P. Arthur, G. Neubig, and S. Nakamura, "Incorporating discrete translation lexicons into neural machine translation," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2016.

[72] C. Dyer, V. Chahuneau, and N. A. Smith, "A simple, fast, and effective reparameterization of ibm model 2," in *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2013, pp. 644–648.

[73] G. Neubig, "lamtram: A toolkit for language and translation modeling using neural networks," http://www.github.com/neubig/lamtram, 2015.

[74] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[75] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting." *Journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[76] H. Hata, O. Mizuno, and T. Kikuno, "Historage: Fine-grained version control system for Java," in *Proc. of 3rd Joint Int. and Annual ERCIM Workshops on Principles of Softw. Evolution and Softw. Evolution Workshops*, ser. IWPSE-EVOL '11. New York, NY, USA: ACM, 2011, pp. 96–100. [Online]. Available: http://doi.acm.org/10.1145/2024445.2024463

[77] K. Fujiwara, H. Hata, E. Makihara, Y. Fujihara, N. Nakayama, H. Iida, and K. Matsumoto, "Kataribe: A hosting service of historage repositories," in *Proc. of 11th Work. Conf. on Mining Softw. Repositories*, ser. MSR '14. New York, NY, USA: ACM, 2014, pp. 380–383. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597125

[78] K. Uemura, Y. Saito, S. Fujiwara, D. Tanaka, K. Fujiwara, H. Iida, and K. Matsumoto, "A hosting service of multi-language historage repositories," in *Proc. of IEEE/ACIS 15th Int. Conf. on Comput. and Inf. Sci. (ICIS 2016)*, June 2016, pp. 1–6.

[79] Y. Sulistyo Nugroho, H. Hata, and K. Matsumoto, "How Different Are Different diff Algorithms in Git? Use –histogram for Code Changes," *arXiv e-prints*, p. arXiv:1902.02467, Feb 2019.

[80] 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 *Proc. of 28th IEEE/ACM Int. Conf. on Automated Softw. Eng.*, ser. ASE'13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 180–190. [Online]. Available: https://doi.org/10.1109/ASE.2013.6693078

[81] A. Alali, H. Kagdi, and J. I. Maletic, "What's a typical commit? a characterization of open source software repositories," in *2008 16th IEEE International Conference on Program Comprehension*, June 2008, pp. 182–191.

[82] O. Arafat and D. Riehle, "The commit size distribution of open source software," in *2009 42nd Hawaii International Conference on System Sciences*, Jan 2009, pp. 1–8.

[83] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725. [Online]. Available: https://www.aclweb.org/anthology/P16-1162

[84] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: Analytics and risk prediction of software commits," in *Proc. of 10th Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 966–969. [Online]. Available: http://doi.acm.org/10.1145/2786805.2803183

[85] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. of 2nd Int. Workshop on Mining Softw. Repositories*, ser. MSR '05. New York, NY, USA: ACM, 2005, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083147

[86] J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg, "Measuring and modeling programming experience," *Empirical Softw. Engg.*, vol. 19, no. 5, pp. 1299–1334, Oct. 2014. [Online]. Available: http://dx.doi.org/10.1007/s10664-013-9286-4

[87] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proc. of 23rd Int. Symp. on Softw. Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440. [Online]. Available: http://doi.acm.org/10.1145/2610384.2628055

[88] S. Chakraborty, M. Allamanis, and B. Ray, "Tree2tree neural translation model for learning source code changes," *CoRR*, vol. abs/1810.00314, 2018. [Online]. Available: http://arxiv.org/abs/1810.00314

[89] W. He, Z. He, H. Wu, and H. Wang, "Improved neural machine translation with smt features," in *Proc. of the 30th AAAI Conf. on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, pp. 151–157. [Online]. Available: http://dl.acm.org/citation.cfm?id=3015812.3015835

[90] X. Wang, Z. Lu, Z. Tu, H. Li, D. Xiong, and M. Zhang, "Neural machine translation advised by statistical machine translation," in *AAAI Conference on Artificial Intelligence*, 2017. [Online]. Available: https://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14451

[91] P. Arthur, G. Neubig, and S. Nakamura, "Incorporating discrete translation lexicons into neural machine translation." in *Proc. of Conference on Empirical Methods in Natural Language Processing*, ser. EMNLP'16, 2016, pp. 1557–1567.

[92] Z. Tu, Y. Liu, L. Shang, X. Liu, and H. Li, "Neural machine translation with reconstruction," in *AAAI Conference on Artificial Intelligence*, 2017. [Online]. Available: https://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14161

[93] L. Liu, M. Utiyama, A. M. Finch, and E. Sumita, "Neural machine translation with supervised attention," *CoRR*, vol. abs/1609.04186, 2016. [Online]. Available: http://arxiv.org/abs/1609.04186

[94] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," *CoRR*, vol. abs/1609.08144, 2016. [Online]. Available: http://arxiv.org/abs/1609.08144

[95] O. Mizuno and T. Kikuno, "Training on errors experiment to detect fault-prone software modules by spam filter," in *Proc. of 6th Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 405–414. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287683

[96] H. Hata, O. Mizuno, and T. Kikuno, "An extension of fault-prone filtering using precise training and a dynamic threshold," in *Proc. of 5th Work. Conf. on Mining Softw. Repositories*, ser. MSR '08. New York, NY, USA: ACM, 2008, pp. 89–98. [Online]. Available: http://doi.acm.org/10.1145/1370750.1370772

[97] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *Proc. of 41st Int. Conf. on Softw. Eng.*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 25–36. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00021

[98] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *CoRR*, vol. abs/1901.01808, 2019. [Online]. Available: http://arxiv.org/abs/1901.01808

[99] S. Saha, R. K. Saha, and M. R. Prasad, "Harnessing evolution for multi-hunk program repair," in *Proc. of 41st Int. Conf. on Softw. Eng.*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 13–24. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00020

[100] S. Kim, E. J. Whitehead, Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Softw. Eng.*, vol. 34, pp. 181–196, March 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1399105.1399451

[101] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *Proc. of 34th Int. Conf. on Softw. Eng.*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 200–210. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337247

[102] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *Proc. of 38th Int. Conf. on Softw. Eng.*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 428–439. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884848

[103] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Trans. Softw. Eng.*, vol. 43, no. 7, pp. 641–657, July 2017.

[104] 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," in *Proc. of 33rd IEEE/ACM Int. Conf. on Automated Softw. Eng.*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 832–837. [Online]. Available: http://doi.acm.org/10.1145/3238147.3240732

[105] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation," *arXiv e-prints*, p. arXiv:1812.08693, Dec 2018.

[106] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A Survey of Machine Learning for Big Code and Naturalness," *ArXiv e-prints*, Sep. 2017.

[107] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *Proc. of 10th Work. Conf. on Mining Softw. Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 207–216. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487127

[108] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proc. of 22nd ACM SIGSOFT Int. Symp. on Found. of Softw. Eng.*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 281–293. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635883

[109] M. Allamanis and C. Sutton, "Mining idioms from source code," in *Proc. of 22nd ACM SIGSOFT Int. Symp. on Found. of Softw. Eng.*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 472–483. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635901

[110] M. Allamanis, D. Tarlow, A. D. Gordon, and Y. Wei, "Bimodal modelling of source code and natural language," in *Proc. of 32nd Int. Conf. on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, pp. 2123–2132. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045118.3045344

[111] P. Bielik, V. Raychev, and M. Vechev, "Phog: Probabilistic model for code," in *Proc. of 33rd Int. Conf. on Machine Learning - Volume 48*, ser. ICML'16. JMLR.org, 2016, pp. 2933–2942. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045390.3045699

[112] J. C. Campbell, A. Hindle, and J. N. Amaral, "Syntax errors just aren't natural: Improving error reporting with language models," in *Proc. of 11th Work. Conf. on Mining Softw. Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 252–261. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597102

[113] L. Cerulo, M. Di Penta, A. Bacchelli, M. Ceccarelli, and G. Canfora, "Irish: A hidden markov model to detect coded information islands in free text," *Sci. Comput. Program.*, vol. 105, no. C, pp. 26–43, Jul. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2014.11.017

[114] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "Synthesizing benchmarks for predictive modeling," in *Proc. of 2017 Int. Symp. on Code Generation and Optimization*, ser. CGO '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 86–99. [Online]. Available: http://dl.acm.org/citation.cfm?id=3049832.3049843

[115] S. Gulwani and M. Marron, "Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation," in *Proc. of 2014 ACM SIGMOD Int. Conf. on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 803–814. [Online]. Available: http://doi.acm.org/10.1145/2588555.2612177

[116] T. Gvero and V. Kuncak, "Synthesizing java expressions from free-form queries," in *Proc. of 2015 ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Syst., Languages, and Appl.*, ser. OOPSLA 2015. New York, NY, USA: ACM, 2015, pp. 416–432. [Online]. Available: http://doi.acm.org/10.1145/2814270.2814295

[117] H. Hata, O. Mizuno, and T. Kikuno, "Fault-prone module detection using large-scale text features based on spam filtering," *Empirical Softw. Eng.*, vol. 15, pp. 147–165, April 2010. [Online]. Available: http://dx.doi.org/10.1007/s10664-009-9117-9

[118] V. J. Hellendoorn, P. T. Devanbu, and A. Bacchelli, "Will they like this?: Evaluating code contributions with language models," in *Proc. of 12th Work. Conf. on Mining Softw. Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 157–167. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820539

[119] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proc. of 11th Joint Meeting on Found. of Softw. Eng.*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 763–773. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106290

[120] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proc. of 34th Int. Conf. on Softw. Eng.*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 837–847. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337322

[121] C.-H. Hsiao, M. Cafarella, and S. Narayanasamy, "Using web corpus statistics for program analysis," in *Procl of 2014 ACM Int. Conf. on Object Oriented Programming Syst. Languages and Appl.*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 49–65. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660226

[122] S. Karaivanov, V. Raychev, and M. Vechev, "Phrase-based statistical translation of programming languages," in *Proc. of 2014 ACM Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2014. New York, NY, USA: ACM, 2014, pp. 173–184. [Online]. Available: http://doi.acm.org/10.1145/2661136.2661148

[123] N. Kushman and R. Barzilay, "Using semantic unification to generate regular expressions from natural language," in *Proc. of 2013 Conf. of North American Chapter of Association for Computational Linguistics: Human Language Technologies*. Atlanta, Georgia: Association for Computational Linguistics, June 2013, pp. 826–836. [Online]. Available: http://www.aclweb.org/anthology/N13-1103

[124] C. J. Maddison and D. Tarlow, "Structured generative models of natural source code," in *Proc. of 31st Int. Conf. on Machine Learning - Volume 32*, ser. ICML'14. JMLR.org, 2014, pp. II–649–II–657. [Online]. Available: http://dl.acm.org/citation.cfm?id=3044805.3044965

[125] A. K. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. T. Kalai, "A machine learning framework for programming by example," in *Proc. of 30th Int. Conf. on Machine Learning - Volume 28*, ser. ICML'13. JMLR.org, 2013, pp. I–187–I–195. [Online]. Available: http://dl.acm.org/citation.cfm?id=3042817.3042840

[126] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proc. of 9th Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 532–542. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491458

[127] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Divide-and-conquer approach for multi-phase statistical migration for source code (t)," in *Proc. of 30th IEEE/ACM Int. Conf. on Automated Softw. Eng.*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 585–596. [Online]. Available: https://doi.org/10.1109/ASE.2015.74

[128] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proc. of 37th Int. Conf. on Softw. Eng. - Vol. 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 858–868. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818858

[129] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Learning api usages from bytecode: A statistical approach," in *Proc. of 38th Int. Conf. on Softw. Eng.*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 416–427. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884873

[130] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proc. of 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 419–428. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594321

[131] V. Raychev, P. Bielik, M. Vechev, and A. Krause, "Learning programs from noisy data," in *Proc. of 43rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: ACM, 2016, pp. 761–774. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837671

[132] A. Sharma, Y. Tian, and D. Lo, "Nirmal: Automatic identification of software relevant tweets leveraging language model," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015, pp. 449–458.

[133] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proc. of 22nd ACM SIGSOFT Int. Symp. on Found. of Softw. Eng.*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 269–280. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635875

[134] B. Vasilescu, C. Casalnuovo, and P. Devanbu, "Recovering clear, natural identifiers from obfuscated js names," in *Proc. of 11th Joint Meeting on Found. of Softw. Eng.*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 683–693. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106289

[135] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proc. of 12th Work. Conf. on Mining Softw. Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 334–345. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820559

[136] S. Yadid and E. Yahav, "Extracting code from programming tutorial videos," in *Proc. of 2016 ACM Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2016. New York, NY, USA: ACM, 2016, pp. 98–111. [Online]. Available: http://doi.acm.org/10.1145/2986012.2986021

[137] S. Kim, K. Pan, and E. E. J. Whitehead, Jr., "Memories of bug fixes," in *Proc. of 14th ACM SIGSOFT Int. Symp. on Found. of Softw. Eng.*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, 2006, pp. 35–45. [Online]. Available: http://doi.acm.org/10.1145/1181775.1181781

[138] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *Proc. of 32nd Int. Conf. on Softw. Eng.*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 315–324. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806847

[139] N. Meng, M. Kim, and K. S. McKinley, "Lase: Locating and applying systematic edits by learning from examples," in *Proc. of 35th Int. Conf. on Softw. Eng.*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 502–511. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486855

[140] R. Yue, N. Meng, and Q. Wang, "A characterization study of repeated bug fixes," in *Proc. of 33rd IEEE Int. Conf. on Softw. Maintenance and Evolution*, Sept 2017, pp. 422–432.