An Empirical Investigation into Learning Bug-Fixing Patches in the Wild via Neural Machine Translation

Michele Tufano
College of William and Mary
Williamsburg, VA, USA

Cody Watson
College of William and Mary
Williamsburg, VA, USA

Gabriele Bavota
Università della Svizzera italiana (USI)
Lugano, Switzerland

Massimiliano Di Penta
University of Sannio
Benevento, Italy

Martin White
College of William and Mary
Williamsburg, VA, USA

Denys Poshyvanyk
College of William and Mary
Williamsburg, VA, USA

ABSTRACT

Millions of open-source projects with numerous bug fixes are available in code repositories. This proliferation of software development histories can be leveraged to learn how to fix common programming bugs. To explore such a potential, we perform an empirical study to assess the feasibility of using Neural Machine Translation techniques for learning bug-fixing patches for real defects. We mine millions of bug-fixes from the change histories of GitHub repositories to extract meaningful examples of such bug-fixes. Then, we abstract the buggy and corresponding fixed code, and use them to train an Encoder-Decoder model able to translate buggy code into its fixed version. Our model is able to fix hundreds of unique buggy methods in the wild. Overall, this model is capable of predicting fixed patches generated by developers in 9% of the cases.

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools*;

KEYWORDS

neural machine translation, bug-fixes

ACM Reference Format:
Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An Empirical Investigation into Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3238147.3240732

1 INTRODUCTION

Localizing and fixing bugs is known to be an effort-prone and time-consuming task for software developers [15, 30, 38]. To support programmers in this activity, researchers have proposed a number of approaches aimed at automatically repairing programs. The proposed techniques either use a generate-and-validate approach, which consists of generating many repairs (*e.g.,* through Genetic Programming like GenProg [23, 37]), or an approach that produces a single fix [14, 27]. While automated program repair techniques still face many challenges to be applied in practice, existing work has made strides to be effective in specific cases. These approaches, given the right circumstances, substantially contribute in reducing the cost of bug-fixes for developers [21, 26].

Two major problems automated repair approaches have are producing patches acceptable for programmers and, especially for generate-and-validate techniques, over-fitting patches to test cases. To cope with this problem, Le *et al.* [20] leverage the past history of existing projects — in terms of bug-fix patches — and compare automatically-generated patches with existing ones. Patches that are similar to the ones found in the past history of mined projects are considered to be more relevant. Another approach that identifies patches from past fixes is Prophet [24], which after having localized the likely faulty code by running test cases, generates patches from correct code using a probabilistic model.

Our work is motivated by the following three considerations. First, automated repair approaches are based on a relatively limited and manually-crafted (with substantial effort and expertise) set of transformations or fixing patterns. Second, the work done by Le *et al.* [20] shows that the past history of existing projects can be successfully leveraged to understand what a "meaningful" program repair patch is. Third, several works have recently demonstrated the capability of advanced machine learning techniques, such as deep learning, to learn from large software engineering (SE) datasets. Some examples of recent models that can be used in a number of SE tasks include: code completion [29], defect prediction [36], bug localization [19], clone detection [39], code search [11], learning API sequences [12], or recommending method names [2].

Forges like GitHub provide a plethora of change history and bug-fixing commits from a large number of software projects. A machine-learning based approach can leverage this data to learn about bug-fixing activities in the wild. In this work, we evaluate the suitability of a Neural-Machine Translation (NMT)-based approach for the task of automatically generating patches for buggy code.

Automatically learning from bug-fixes in the wild provides the ability to emulate real patches written by developers. Additionally, we harness the power of NMT to "translate" buggy code into fixed code thereby emulating the combination of AST operations performed in the developer written patches. Further benefits include the static nature of NMT when identifying candidate patches, since, unlike some generate-and-validate approaches, we do not need to execute test cases during patch generation [31, 40].

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3240732

We start by mining a large set of (~ 787k) bug-fixing commits from GitHub. From these commits, we extract method-level AST edit operations using fine-grained source code differencing [7]. We identify multiple method-level differences per bug-fixing commit and independently consider each one, yielding to ~ 2.3M bug-fix pairs (BFPs). After that, the code of the BFPs is abstracted to make it more suitable for the NMT model. Finally, an encoder-decoder model is used to understand how the buggy code is transformed into fixed code. Once the model has been trained, it is used to generate patches for unseen code.

We empirically investigate the potential of NMT to generate candidate patches that are identical to the ones implemented by developers. The results indicate that trained NMT model is able to successfully predict the fixed code, given the buggy code, in 9% of the cases.

2 APPROACH

Fig. 1 shows an overview of the NMT approach that we experiment with. The black boxes represent the main phases, the arrows indicate data flows, and the dashed arrows denote dependencies on external tools or data. We mine bug-fixing commits from thousands of GitHub repos using GitHub Archive [10] (Sec. 2.1). From the bug-fixes, we extract method-level pairs of *buggy* and corresponding *fixed* code named *bug-fix pairs* (BFPs) (Sec. 2.2.1). BFPs are the examples that we use to learn how to fix code from bug-fixes (*buggy* $\rightarrow$ *fixed*). We use GumTree [7] to identify the list of edit actions (A) performed between the buggy and fixed code. Then, we use a Java Lexer and Parser to abstract the source code of the BFPs (Sec. 2.2.2) into a representation better suited for learning. During the abstraction, we keep frequent identifiers and literals we call *idioms* within the representation. The output of this phase are the abstracted BFPs and their corresponding mapping M, which allows reconstructing the original source code. Next, we filter out long methods (Sec. 2.2.3) and we use the obtained set to train an encoder-decoder model able to learn how to transform a *buggy* code into a corresponding *fixed* version (Sec. 2.3). The trained model can be used to generate a patch for unseen buggy code.

2.1 Bug-Fixes Mining

We downloaded from GitHub Archive [10] every public GitHub event between March 2011 and October 2017 and we used the Google BigQuery APIs to identify all commits having a message containing the patterns [8]: ("fix" or "solve") and ("bug" or "issue" or "problem" or "error"). We identified ~10M (10,056,052) bug-fixing commits. As the content of commit messages and issue trackers might imprecisely identify bug-fixing commits, two authors independently analyzed a statistically significant sample (95% confidence level ±5% confidence interval, for a total size of 384) of identified commits to check whether they were actually bug fixes. After solving 13 cases of disagreement, they concluded that 97.6% of the identified bug-fixing commits were true positive. Details about this evaluation are in our online appendix [34].

For each bug-fixing commit, we extracted the source code before and after the bug-fix using the GitHub Compare API [9]. This allowed us to collect the buggy (pre-commit) and the fixed (post-commit) code. We discarded commits related to non-Java files, as well as files that were created in the bug-fixing commit, since there would be no buggy version to learn from. Moreover, we discarded commits impacting more than five Java files, since we aim to learn focused bug-fixes that are not spread across the system. The result of this process was the buggy and fixed code of 787,178 bug-fixing commits.

2.2 Bug-Fix Pairs Analysis

A BFP (Bug-Fixing Pair) is a pair (m_b, m_f) where m_b represents a buggy code component and m_f represents the corresponding fixed code. We will use these BFPs to train the NMT model, make it learning the translation from buggy (m_b) to fixed (m_f) code, thus being able of generating patches.

2.2.1 Extraction. Given (f_b, f_f) a pair of buggy and fixed file from a bug-fix bf, we used the GumTree Spoon AST Diff [7] tool to compute the AST differencing between f_b and f_f. This computes the sequence of edit actions performed at the AST level that allows to transform the f_b's AST into the f_f's AST.

Since the file-level granularity could be too large to learn patterns of transformation, we separate the code into method-level fragments that will constitute our BFPs. The rationale for choosing method-level BFPs is supported by several reasons. First, methods represent a reasonable target for fixing activities, since they are likely to implement a single task or functionality. Second, methods provide enough meaningful context for learning fixes, such as variables, parameters, and method calls used in the method. This choice is justified by recent empirical studies, which indicated how the large majority of fixing patches consist of single line, single churn or, worst cases, churns separated by a single line [32]. Smaller snippets of code lack the necessary context and, hence, they could not be considered. Finally, considering arbitrarily long snippets of code, such as hunks in diffs, makes learning more difficult given the variability in size and context [1, 18].

We first rely on GumTree to establish the mapping between the nodes of f_b and f_f. Then, we extract the list of mapped pairs of methods $L = \{(m_{1b}, m_{1f}), \ldots, (m_{nb}, m_{nf})\}$. Each pair (m_{ib}, m_{if}) contains the method m_{ib} (from the buggy file f_b) and the corresponding method m_if (from the fixed file f_f). Next, for each pair of mapped methods, we extract a sequence of edit actions using the GumTree algorithm. We then consider only those method pairs for which there is at least one edit action (*i.e.,* we disregard methods that have not been modified during the fix). Therefore, the output of this phase is a list of $BFPs = \{bfp_1, \ldots, bfp_k\}$, where each BFP is a triplet $bfp = \{m_b, m_f, A\}$, where m_b is the buggy method, m_f is the corresponding fixed method, and A is a sequence of edit actions that transforms m_b in m_f. We exclude methods created/deleted during the fixing, since we cannot learn fixing operations from them. Overall, we extracted ~2.3M BFPs.

It should be noted that the process we use to extract the BFPs: (i) does not capture changes performed outside methods (*e.g.,* class signature, attributes, *etc.*), and (ii) considers each BFP as an independent bug fix, meaning that multiple methods modified in the same bug fixing activity are considered independently from one another.

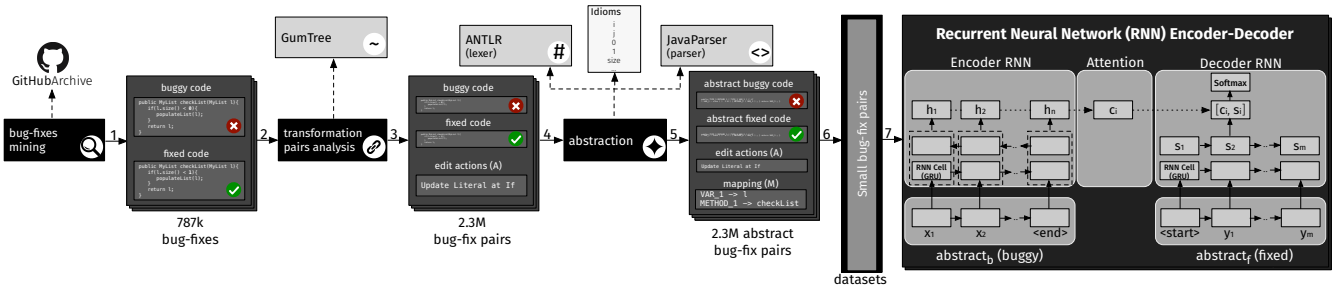

Figure 1: Overview of the process used to experiment with an NMT-based approach.

2.2.2 Abstraction. Learning bug-fixing patterns is extremely challenging by working at the level of raw source code. This is especially due to the huge vocabulary of terms used in the identifiers and literals of the ~2M mined projects. Such a large vocabulary would hinder our goal of learning transformations of code as a neural machine translation task. For this reason, we abstract the code and generate an expressive yet vocabulary-limited representation. We use a Java lexer and a parser to represent each buggy and fixed method within a BFP as a stream of tokens. The lexer, built on top of ANTLR [28], tokenizes the raw code into a stream of tokens, that is then fed into a Java parser [35], which discerns the role of each identifier (*i.e.,* whether it represents a variable, method, or type name) and the type of a literal.

Each BFP is abstracted in isolation. Given a BFP $bfp = \{m_b, m_f, A\}$, we first consider the source code of m_b. The source code is fed to a Java lexer, producing the stream of tokens. The stream of tokens is then fed to a Java parser, which recognizes the identifiers and literals in the stream. The parser generates and substitutes a unique ID for each identifier/literal within the tokenized stream. If an identifier or literal appears multiple times in the stream, it will be replaced with the same ID. The mapping of identifiers/literals with their corresponding IDs is saved in a map (M). The final output of the Java parser is the abstracted method ($abstract_b$). Then, we consider the source code of m_f. The Java lexer produces a stream of tokens, which is then fed to the parser. The parser continues to use a map M when abstracting m_f. The parser generates new IDs only for novel identifiers/literals, not already contained in M, meaning, they exist in m_f but not in m_b. Then, it replaces all the identifiers/literals with the corresponding IDs, generating the abstracted method ($abstract_f$). The abstracted BFP is now a 4-tuple $bfpa = \{abstract_b, abstract_f, A, M\}$, where M is the ID mapping for that particular BFP. The process continues considering the next BFP, generating a new mapping M. Note that we first analyze the buggy code m_b and then the corresponding fixed code m_f of a BFP, since this is the direction of the learning process.

IDs are assigned to identifiers and literals in a sequential and positional fashion: The first method name found will be assigned the ID of `METHOD_1`, likewise the second method name will receive the ID of `METHOD_2`. This process continues for all the method and variable names (`VAR_X`) as well as the literals (`STRING_X`, `INT_X`, `FLOAT_X`).

At this point, $abstract_b$ and $abstract_f$ of a BFP are a stream of tokens consisting of language keywords (*e.g.,* `for`, `if`), separators (*e.g.,* "(", ";", "}") and IDs representing identifiers and literals. Comments and annotations have been removed from the code representation.

Some identifiers and literals appear so often in the code that, for the purpose of our abstraction, they can almost be treated as keywords of the language. This is the case for the variables `i`, `j`, or `index`, that are often used in loops, or for literals such as `0`, `1`, `-1`, often used in conditional statements and return values. Similarly, method names, such as `size` or `add`, appear several times in our code base, since they represent common concepts. These identifiers and literals are often referred to as "idioms" [5]. We include idioms in our representation and do not replace idioms with a generated ID, but rather keep the original text when abstracting the code.

To define the list of idioms, we first randomly sampled 300k BFPs and considered all their original source code. Then, we extracted the frequency of each identifier/literal used in the code, discarding keywords, separators, and comments. Next, we analyzed the distribution of the frequencies and focused on the top 0.005% frequent words (outliers of the distribution). Two authors manually analyzed this list and curated a set of 272 idioms also including standard Java types such as `String`, `Integer`, common `Exceptions`, *etc.* The list of idioms is available in the online appendix [34].

This representation provides enough context and information to effectively learn code transformations, while keeping a limited vocabulary ($|V| = \sim 430$). The abstracted code can be mapped back to the real source code using the mapping (M).

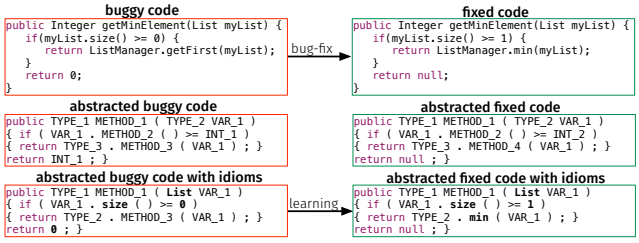

Figure 2: Code Abstraction Example.

To better understand our representation, let us consider the example in Fig. 2, where we see a bug-fix related to finding the minimum value in a list of integers. The buggy method contains three errors, which the fixed code rectifies. The first bug is within

the if-condition, where the buggy method checks if the list size is greater than or equal to `0`. This is problematic since a list without any values cannot have a minimum value to return. The second bug is in the method call `getFirst`, this will return the first element in the list, which may or may not be the minimum value. Lastly, if the if-condition fails in the buggy method then the method returns `0`; returning `0` when the minimum is unable to be identified is incorrect as it indicates that one of the elements within the list is `0`. The fixed code changes the if-condition to compare against a list size of `1` rather than `0`, uses the `min` method to return the minimum value and changes the return value to `null` when the if-condition fails.

Using the buggy and fixed code for training, although a viable and realistic bug-fix, presents some issues. When we feed the buggy piece of code to the Java Parser and Lexer, we identify some problems with the mapping. For example, the abstracted fixed code contains `INT_2` and `METHOD_4`, which are not contained in the abstracted version of the buggy code or its mapping. Since the mapping of tokens to code is solely reliant on the buggy method, this example would require the synthesis of new values for `INT_2` and `METHOD_4`. However, the methodology takes advantage of idioms, allowing to still consider this BFP. When using the abstraction with idioms, we are able to replace tokens with the values they represent. Now, when looking at the abstracted code with idioms for both buggy and fixed code, there are no abstract tokens found in the fixed code that are not in the buggy code. Previously, we needed to synthesize values for `INT_2` and `METHOD_4`, however, `INT_2` was replaced with idiom `1` and `METHOD_4` with idiom `min`. With the use of idioms, we are capable of keeping this BFP while maintaining the integrity of learning real, developer inspired patches.

2.2.3 Filtering. We filter out BFPs that: (i) contain lexical or syntactic errors (*i.e.,* either the lexer or parser fails to process them) in either the buggy or fixed code; (ii) their buggy and fixed abstracted code ($abstract_b$, $abstract_f$) resulted in equal strings; (iii) performed more than 100 atomic AST actions ($|A| > 100$) between the buggy and fixed version. The rationale behind the latter decision was to eliminate outliers of the distribution (the 3rd quartile of the distribution is 14 actions), which could hinder the learning process. Moreover, we do not aim to learn such large bug-fixing patches.

Next, we filter the BFPs based on their size, measured in the number of tokens. We decided to disregard long methods (longer than 50 tokens) and focused on small size BFPs. We, therefore, create the dataset $BFP_{small} = \{bfp \leq 50\}$.

2.2.4 Synthesis of Identifiers and Literals. BFPs are the examples we use to make our model learn how to fix source code. Given a $bfp = \{m_b, m_f, A\}$, we first abstract its code, obtaining $bfpa = \{abstract_b, abstract_f, A, M\}$. The buggy code $abstract_b$ is used as input to the model, which is trained to output the corresponding fixed code $abstract_f$. This output can then be mapped back to real source code using M.

In the real usage scenario, when the model is deployed, we do not have access to the oracle (*i.e.,* fixed code, $abstract_f$), but only to the input code. This source code can then be abstracted and fed to the model, which generates as output a predicted code ($abstract_p$). The IDs that the $abstract_p$ contains can be mapped back to real values only if they also appear in the input code. If the fixed code suggests to introduce a method call, `METHOD_6`, which is not found in the input code, we cannot automatically map `METHOD_6` to an actual method name. This inability to map back source code exists for any newly created ID generated for identifiers or literals, which are absent in the input code.

Therefore, it appears that the abstraction process, which allows us to limit the vocabulary size and facilitate the training process, confines us to only learning fixes that re-arrange keywords, identifiers, and literals already available in the context of the buggy method. This is the primary reason we decided to incorporate idioms in our code representation, and treat them as keywords of the language. Idioms help retaining BFPs that otherwise would be discarded because of the inability to synthesize new identifiers or literals. This allows the model to learn how to replace an abstract identifier/literal with an idiom or an idiom with another idiom (*e.g.,* bottom part of Fig. 2).

After these filtering phases, the datasets BFP_{small} is comprised of 58k (58,350) bug-fixes.

2.3 Learning Patches

2.3.1 Dataset Preparation. Given a set of BFPs (*i.e.,* BFP_{small}) we use the instances to train an Encoder-Decoder model. Given a $bfpa = \{abstract_b, abstract_f, A, M\}$ we use only the pair ($abstract_b$, $abstract_f$) of buggy and fixed abstracted code for learning. No additional information about the possible fixing actions (A) is provided during the learning process to the model. The given set of BFPs is randomly partitioned into: training (80%), validation (10%), and test (10%) sets. Before the partitioning, we make sure to remove any duplicated pairs ($abstract_f$, $abstract_b$) to not bias the results, *i.e.,* same pair both in training and test set.

2.3.2 NMT. The experimented model is based on an RNN Encoder-Decoder architecture, commonly adopted in NMT [6, 17, 33]. This model consists of two major components: an RNN Encoder, which *encodes* a sequence of terms $\boldsymbol{x}$ into a vector representation, and an RNN Decoder, which *decodes* the representation into another sequence of terms $\boldsymbol{y}$. The model learns a conditional distribution over a (output) sequence conditioned on another (input) sequence of terms: $P(y_1, .., y_m | x_1, .., x_n)$, where n and m may differ. In our case, given an input sequence $\mathbf{x} = abstract_b = (x_1, .., x_n)$ and a target sequence $\mathbf{y} = abstract_f = (y_1, .., y_m)$, the model is trained to learn the conditional distribution: $P(abstract_f | abstract_b) = P(y_1, .., y_m | x_1, .., x_n)$, where x_i and y_j are abstracted source tokens: Java keywords, separators, IDs, and idioms. Fig. 1 shows the architecture of the Encoder-Decoder model with attention mechanism [3, 4, 25]. The Encoder takes as input a sequence $\mathbf{x} = (x_1, .., x_n)$ and produces a sequence of states $\mathbf{h} = (h_1, .., h_n)$. We rely on a bi-directional RNN Encoder [3], which is formed by a backward and a forward RNN, which are able to create representations taking into account both past and future inputs [4]. That is, each state h_i represents the concatenation (dashed box in Fig. 1) of the states produced by the two RNNs when reading the sequence in a forward and backward fashion: $h_i = [\overrightarrow{h_i}; \overleftarrow{h_i}]$.

The RNN Decoder predicts the probability of a target sequence $\mathbf{y} = (y_1, .., y_m)$ given $\mathbf{h}$. Specifically, the probability of each output term y_i is computed based on: (i) the recurrent state s_i in the Decoder; (ii) the previous $i-1$ terms $(y_1, .., y_{i-1})$; and (iii) a context vector c_i. The latter constitutes the attention mechanism. The

vector c_i is computed as a weighted average of the states in $\mathbf{h}$: $c_i = \sum_{t=1}^{n} a_{it} h_t$ where the weights a_{it} allow the model to pay more *attention* to different parts of the input sequence. Specifically, the weight a_{it} defines how much the term x_i should be taken into account when predicting the target term y_t.

The entire model is trained end-to-end (Encoder and Decoder jointly) by minimizing the negative log likelihood of the target terms, using stochastic gradient descent.

2.3.3 Hyperparameter Search. For the model built on the BFP_{small} dataset (*i.e.,* M_{small}) we performed hyperparameter search by testing ten configurations of the encoder-decoder architecture. The configurations tested different combinations of RNN Cells (LSTM [13] and GRU [6]), number of layers (1, 2, 4) and units (256, 512) for the encoder/decoder, and the embedding size (256, 512). Bucketing and padding was used to deal with the variable length of the sequences. We trained our models for a maximum of 60k epochs, and selected the model's checkpoint before over-fitting the training data. To guide the selection of the best configuration, we used the loss function computed on the *validation* set (not on the test set), while the results are computed on the *test* set. All data are available in our online appendix [34].

3 EXPERIMENTAL DESIGN

The *goal* of this study is to empirically assess whether NMT can be used to learn fixes in the wild. The *context* consists of a dataset of bug fixes (Sec. Section 2) and aims at answering the following research question.

3.1 RQ: Is Neural Machine Translation a viable approach to learn how to fix code?

We aim to empirically assessing whether NMT is a viable approach to learn transformations of the code from a buggy to a fixed state. To this end, we use the dataset BFP_{small} to train and evaluate the NMT model M_{small}. Precisely, given a BFP dataset, we train different configurations of the Encoder-Decoder models, then select the best performing configuration on the validation set. We then evaluate the validity of the model with the unseen instances of the test set.

The evaluation is performed as follows: let M be the trained model and T be the test set of BFPs (BFP_{small}), we evaluate the model M for each $bfp = (abstract_b, abstract_f) \in T$. Specifically, we feed the buggy code $abstract_b$ to the model M, which will generate a single potential patch $abstract_p$. We say that the model generated a successful fix for the code if and only if $abstract_p = abstract_f$. We report the raw count and percentage of successfully fixed BFPs in the test set.

4 RESULTS

4.1 RQ: Is Neural Machine Translation a viable approach to learn how to fix code?

When performing the hyperparameter search, we found that the configuration which achieved the best results on the validation set was the one with 1-layer bi-directional Encoder, 2-layer Attention Decoder both with 256 units, embedding size of 512, and LSTM [13] RNN cells. We trained the M_{small} model for 50k epochs.

The model was able to successfully generate a fix for 538 out of 5,835 cases (**9.22**% of the BFPs in the test set) by "translating" the buggy code in the corresponding fixed code. While the number of successful fixes might appear relatively small, it is important to note that these fixes are generated with a single *guess* of the model as opposed to previous approaches that generate many potential patches. Moreover, it is worth noting that all BFPs in the test sets are unique and have never been seen before by the model during the training or validation steps. All the patches generated by the model can be mapped to *concrete* source code by replacing the IDs in the abstract code to the actual identifiers and literals values stored in the mapping M.

5 THREATS TO VALIDITY

Construct validity. To have enough training data, we mined bug-fixes in GitHub repositories rather than using curated bug-fix datasets such as Defects4j [16] or IntroClass[22], useful but very limited in size. To mitigate imprecisions in our datasets, we manually analyzed a sample of the extracted commits and verified that they were related to bug-fixes.

Internal. It is possible that the performance of our model depends on the hyperparameter configuration. We explain in Section 2.3.3 how hyperparameter search has been performed.

External. We did not compare NMT models with state-of-the-art techniques supporting automatic program repair since our main goal was not to propose a novel approach for automated program repair, but rather to execute a large-scale empirical study investigating the suitability of NMT for generating patches. Additional steps are needed to convert the methodology we adopted into an end-to-end working tool, such as the automatic implementation of the patch, the running of the test cases checking its suitability, *etc.* This is a part of our future work agenda.

We only focused on Java programs. However, the learning process is language-independent and the whole infrastructure can be instantiated for different programming languages by replacing the lexer, parser and AST differencing tools.

We only focused on small-sized methods. We reached this decision after analyzing the distribution of the extracted BFPs, balancing the amount of training data available and the variability in sentence length.

6 CONCLUSIONS

We presented an empirical investigation into the applicability of NMT for the purpose of learning how to fix code, from real bug-fixes. We first devised and detailed a process to mine, extract, and abstract the source code of bug-fixes available in the wild, in order to obtain method level examples of bug-fixes, which we call BFP. Then, we set up, trained, and tuned NMT models to *translate* buggy code into fixed code. We found that our model is able to fix a large number of unique bug-fixes, accounting for 9% of the used BFPs.

This study constitutes a solid empirical foundation upon which other researchers could build, and appropriately evaluate, program repair techniques based on NMT.

REFERENCES

[1] Abdulkareem Alali, Huzefa H. Kagdi, and Jonathan I. Maletic. 2008. What's a Typical Commit? A Characterization of Open Source Software Repositories. In

The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008. 182–191.

[2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 38–49.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. *CoRR* abs/1409.0473 (2014).

[4] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc V. Le. 2017. Massive Exploration of Neural Machine Translation Architectures. *CoRR* abs/1703.03906 (2017).

[5] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. 2017. The Care and Feeding of Wild-caught Mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 511–522. DOI: http://dx.doi.org/10.1145/3106237.3106280

[6] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR* abs/1406.1078 (2014).

[7] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 313–324.

[8] Michael Fischer, Martin Pinzger, and Harald C. Gall. 2003. Populating a Release History Database from Version Control and Bug Tracking Systems. In *19th International Conference on Software Maintenance (ICSM 2003), The Architecture of Existing Systems, 22-26 September 2003, Amsterdam, The Netherlands*. 23. DOI: http://dx.doi.org/10.1109/ICSM.2003.1235403

[9] GitHub. 2010. GitHub Compare API. https://developer.github.com/v3/repos/commits/#compare-two-commits. (2010).

[10] Ilya Grigorik. 2012. GitHub Archive. https://www.githubarchive.org. (2012).

[11] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep Code Search. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 3, 2018*.

[12] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 631–642.

[13] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. DOI: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[14] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated Atomicity-violation Fixing. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 389–400.

[15] Magne Jorgensen and Martin Shepperd. 2007. A Systematic Review of Software Development Cost Estimation Studies. *IEEE Trans. Softw. Eng.* 33, 1 (Jan. 2007), 33–53.

[16] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 437–440.

[17] Nal Kalchbrenner and Phil Blunsom. 2013. Recurrent Continuous Translation Models. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Seattle, Washington, USA, 1700–1709.

[18] Carsten Kolassa, Dirk Riehle, and Michel A. Salim. 2013. A Model of the Commit Size Distribution of Open Source. In *SOFSEM 2013: Theory and Practice of Computer Science*, Peter van Emde Boas, Frans C. A. Groen, Giuseppe F. Italiano, Jerzy Nawrocki, and Harald Sack (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–66.

[19] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2017. Bug localization with combination of deep learning and information retrieval. In *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*. 218–229.

[20] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. 213–224.

[21] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 3–13.

[22] C. Le Goues, N. Holtschulte, E. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *TSE* 41, 12 (2015), 1236–1256.

[23] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.* 38, 1 (2012), 54–72.

[24] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 298–312.

[25] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. *CoRR* abs/1508.04025 (2015).

[26] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering* 22, 4 (2017), 1936–1964.

[27] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 772–781.

[28] Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf.

[29] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 419–428.

[30] Robert C. seacord, Daniel Plakosh, and Grace A. Lewis. 2003. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[31] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 532–543.

[32] Victor Sobreira, Thomas Durieux, Fernanda Madeiral Delfim, Martin Monperrus, and Marcelo de Almeida Maia. 2018. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*. 130–140.

[33] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. *CoRR* abs/1409.3215 (2014).

[34] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Learning Bug-Fixing Patches in the Wild via Neural Machine Translation - Online Appendix. https://sites.google.com/view/learning-fixes. (2018).

[35] Danny van Bruggen. 2014. JavaParser. https://javaparser.org/about.html. (2014).

[36] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 297–308.

[37] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. 364–374.

[38] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. 2007. How Long Will It Take to Fix This Bug?. In *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR '07)*. IEEE Computer Society, Washington, DC, USA, 1–.

[39] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. 87–98.

[40] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better Test Cases for Better Automated Program Repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 831–841. DOI: http://dx.doi.org/10.1145/3106237.3106274