An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation

MICHELE TUFANO, College of William and Mary, USA
CODY WATSON, College of William and Mary, USA
GABRIELE BAVOTA, Università della Svizzera italiana (USI), Switzerland
MASSIMILIANO DI PENTA, University of Sannio, Italy
MARTIN WHITE, College of William and Mary, USA
DENYS POSHYVANYK, College of William and Mary, USA

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. First, we mine millions of bug-fixes from the change histories of projects hosted on GitHub, in order to extract meaningful examples of such bug-fixes. Next, 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. In our empirical investigation we found that such a model is able to fix thousands of unique buggy methods in the wild. Overall, this model is capable of predicting fixed patches generated by developers in 9-50% of the cases, depending on the number of candidate patches we allow it to generate. Also, the model is able to emulate a variety of different Abstract Syntax Tree operations and generate candidate patches in a split second.

CCS Concepts: • **Software and its engineering** → *Maintaining software.*

Additional Key Words and Phrases: neural machine translation, bug-fixes

1 INTRODUCTION

Localizing and fixing bugs is known to be an effort-prone and time-consuming task for software developers [33, 72, 89]. To support programmers in this common activity, researchers have proposed a number of approaches aimed at automatically repairing programs [5, 8, 20, 21, 36, 37, 41, 42, 45, 48, 52, 53, 61, 64, 66, 76, 78, 86, 88, 93]. The proposed techniques either use a generate-and-validate approach, which consists of generating many repairs (*e.g.,* through Genetic Programming like GenProg [45, 87]), or an approach that produces a single fix [32, 61]. While automated program

Authors' addresses: Michele Tufano, College of William and Mary, Williamsburg, Virginia, USA, mtufano@cs.wm.edu; Cody Watson, College of William and Mary, Williamsburg, Virginia, USA, cawatson@cs.wm.edu; Gabriele Bavota, Università della Svizzera italiana (USI), Lugano, Switzerland, gabriele.bavota@usi.ch; Massimiliano Di Penta, University of Sannio, Benevento, Italy, dipenta@unisannio.it; Martin White, College of William and Mary, Williamsburg, Virginia, USA, mgwhite@cs.wm.edu; Denys Poshyvanyk, College of William and Mary, Williamsburg, Virginia, USA, denys@cs.wm.edu.

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 [43, 50].

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. Qi *et al.* [67] found that the majority of the reported patches generated by several generate-and-validate techniques are not correct, and that such techniques mostly achieve repair by deleting pieces of functionality or by overfitting on test cases. To cope with this problem, Le *et al.* [42] leverages the past history of existing projects — in terms of bug-fix patches — and compares 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 [48], 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 required) set of transformations or fixing patterns. Second, the work done by Le *et al.* [42] 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 relatively large software engineering datasets. Some examples of recent models that can be used in a number of software engineering tasks include: code completion [68, 92], defect prediction [85], bug localization [40], clone detection [81, 91], code search [24], learning API sequences [25], recommending method names [3], learning code changes [80] or generating Android APKs from designer's sketches [56].

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 expand upon our original idea of learning bug-fixes [82] and extensively evaluate the suitability of a Neural-Machine Translation (NMT-based approach) to automatically generate 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 Abstract Syntax Tree (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[74, 94]. Test case execution on the patches recommended by the NMT approach would still be necessary in practice, however, this would only be needed on the candidate set of patches.

To this aim, we first mine a large set of ($\sim 787k$) bug-fixing commits from GitHub. From these commits, we extract method-level AST edit operations using fine-grained source code differencing [16]. We identify multiple method-level differences per bug-fixing commit and independently consider each one, yielding to $\sim 2.3M$ 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. Also, we quantitatively and qualitatively analyze the AST
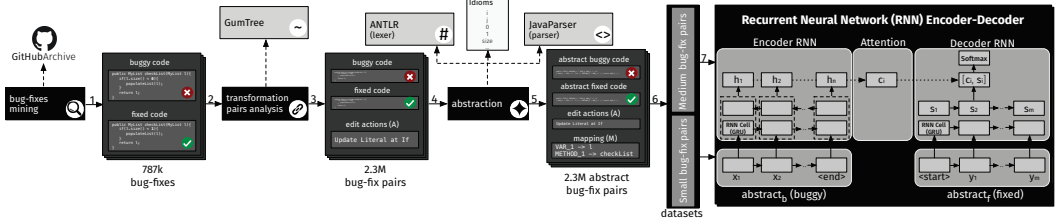
Fig. 1. Overview of the process used to experiment with an NMT-based approach.

operations the NMT model is able to emulate when fixing bugs. Finally, we evaluate its efficiency by computing the time needed to learn a model and to infer patches.

The results indicate that trained NMT models are able to successfully predict the fixed code, given the buggy code, in 9-50% of the cases. The percentage of bugs that can be fixed depends on the number of candidate patches the model is required to produce. We find that over 82% of the generated candidate patches are syntactically correct. When performing the *translation* the models emulate between 27-64% of the AST operations used by developers to fix the bugs, during patch generation. The NMT models are capable of producing multiple candidate patches for a given buggy code in less then a second.

In all, the paper provides the following contributions:

- An extensive empirical investigation into the applicability of NMT techniques for learning how to generate patches from bug-fixes;
- A detailed process for training and evaluating NMT models by mining, extracting, and abstracting bug-fixing examples in the wild;
- A publicly available replication package, including datasets, source code, tools, and detailed results reported and discussed in this study [83].

2 APPROACH

Fig. 1 shows an overview of the NMT approach that we experiment with. The dark 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 repositories using GitHub Archive [23] (Section 2.1). From the bug-fixes, we extract method-level pairs of *buggy* and corresponding *fixed* code named *bug-fix pairs* (BFPs) (Section 2.2.1). BFPs are the examples that we use to learn how to fix code from bug-fixes (*buggy* → *fixed*). We use GumTree [16] 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 (Section 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 generate two datasets of BFPs grouping together fixes for small and medium methods, respectively (Section 2.2.3). Finally, for each set, we use an encoder-decoder model to learn how to transform a *buggy* code into a corresponding *fixed* version (Section 2.3). The trained models can be used to generate patches for unseen buggy code.

2.1 Bug-Fixes Mining

We downloaded from GitHub Archive [23] 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 [17]: ("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 [4, 29], 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 [83].

For each bug-fixing commit, we extracted the source code before and after the bug-fix using the GitHub Compare API [19]. 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 tool [16] 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.

GumTree Diff considers the following edit actions: (i) *updatedValue*: replaces the value of a node in the AST (<Update, AST_Node_Type, Target_AST_Node_Type>); (ii) *add/insert*: inserts a new node in the AST (<Insert, AST_Node_Type, Target_AST_Node_Type>); (iii) *delete*: which deletes a node in the AST (<Delete, AST_Node_Type, Target_AST_Node_Type>); (iv) *move*: moves an existing node in a different location in the AST (<Move, AST_Node_Type, Source_AST_Node_Type, Target_AST_Node_Type>). In our analysis we consider the set of AST edit actions as defined by GumTree Diff.

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 [75].

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, 39].

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 $m_i f$ (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.

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 NMT 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 [62, 63], tokenizes the raw code into a stream of tokens, that is then fed into a Java parser [84], 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" [11]. 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 [83].

This representation provides enough context and information to effectively learn code transformations, while keeping a limited vocabulary ($|V| = \sim430$). The abstracted code can be mapped back to the real source code using the mapping (M).
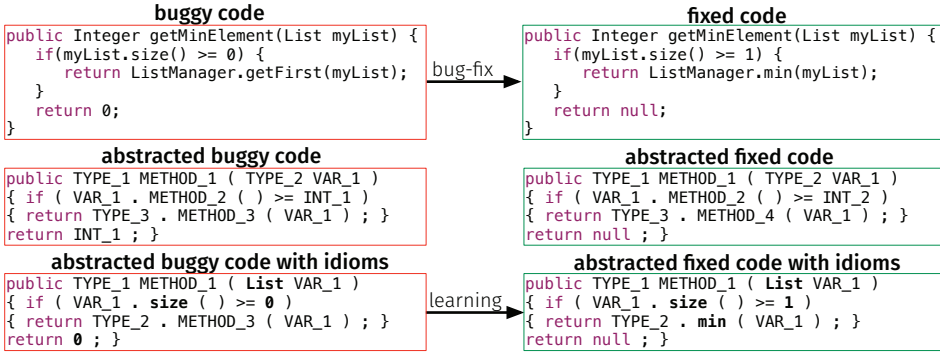

Fig. 2. Code Abstraction Example.

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 called 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 analyze the distribution of BFPs based on their size, measured in the number of tokens, shown in Fig. 3. We can notice that the density of BFPs for the buggy code has a peak before 50 tokens and a long tail that extends over 300 tokens.
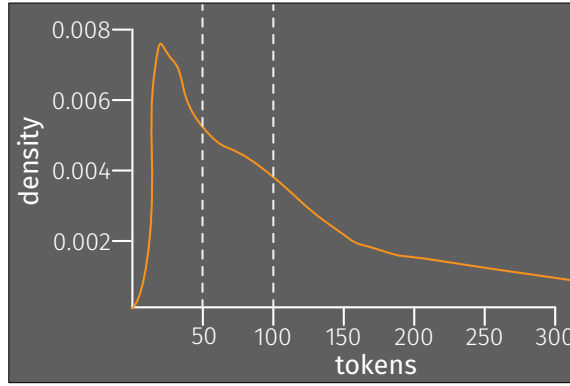

Fig. 3. Distribution of BFPs by the number of tokens.

NMT models require large training dataset in order to achieve reasonable results. Moreover, the variability in sentences length can affect training and performance of the models, even when techniques such as bucketing and padding are employed. For these reasons, we decided to focus on the intervals where most of the data points are available. From Fig. 3 it is clear that most of the data points are concentrated in the interval 0-100. Further analysis showed that there are more data points in the interval 0-100 than in the larger interval 100-500. Therefore, we disregard long methods (longer than 100 tokens) and focused on small/medium size BFPs. We create two datasets: $BFP_{small} = \{bfp \leq 50\}$ and $BFP_{medium} = \{50 < bfp \leq 100\}$.

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 two datasets BFP_{small} and BFP_{medium} consist of 58k (58,350) and 65k (65,455) bug-fixes, respectively.

2.3 Learning Patches

2.3.1 Dataset Preparation. Given a set of BFPs (*i.e.,* BFP_{small} and BFP_{medium}) 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. Specifically, code duplication represents a significant threat to machine learning approaches on source code which could lead to inflated results, as recently described by Allamanis [2]. In particular, there exists no pair $< bfp_i, bfp_j >$ in our dataset (*i.e.,* train, validation, and test combined) such that $bfp_i = bfp_j$. The duplicates are removed after the abstraction process, allowing us not only to discard instances with identical source code, but also those with similar code that becomes identical after the abstraction. As a matter of fact, by replacing identifiers and literals with IDs such as VAR and TYPE, we perform a process similar to the one applied by clone detection tools to identify similar code. Therefore, we go beyond simply removing instances having the same source code.

In order to better check against the presence of clones, we also employ a state-of-the-art clone detection tool, NiCad [69]. NiCad provides pre-defined configurations for different types of clones, which allow us to avoid to set an arbitrary similarity threshold. Also, recent work [70] showed the superiority of NiCad to alternative tools in terms of precision/recall. The only comparable tool is SourcererCC [70], which is above all better in terms of scalability, something not really relevant for our application scenario.

We found no Type I clones, and only ~1,200 and ~400 Type II clone pairs in BFP_{small} and BFP_{medium}, respectively. These few Type II clones come from the idiomatic abstraction process, where instances differ for idioms in the abstract code. Note that, even if these instances represent clones, they are effectively two different inputs/outputs for the model, since their sequences of tokens are different.

2.3.2 NMT. The experimented models are based on an RNN Encoder-Decoder architecture, commonly adopted in NMT [15, 35, 77]. This model consists of two major components: an RNN Encoder, which *encodes* a sequence of terms x into a vector representation, and an RNN Decoder, which *decodes* the representation into another sequence of terms 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 [6, 10, 49]. 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 [6], which is formed by a backward and a forward RNN, which are able to create representations taking
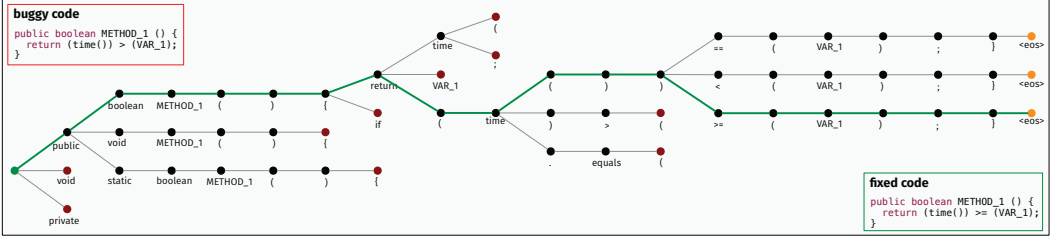
Fig. 4. Beam Search Visualization.

into account both past and future inputs [10]. 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 Generating Multiple Patches via Beam Search. After the model is trained, it is evaluated against the test set of *unseen* buggy code. The classic greedy decoding selects, at each time step i, the output term y_i with the highest probability. The downside of this decoding strategy is that, given a buggy code as input, the trained model will generate only one possible sequence of predicted fixed code. Conversely, we would like to generate multiple potential patches (*i.e.,* sequence of terms representing the fixed code) for a given buggy code. To this aim, we employ a different decoding strategy called Beam Search and used in previous applications of deep learning [6, 9, 22, 68].

The major intuition behind Beam Search decoding is that rather than predicting at each time step the token with the best probability, the decoding process keeps track of k hypotheses (with k being the beam size or width). Formally, let $\mathcal{H}_t$ be the set of k hypotheses decoded till time step t:

$$\mathcal{H}_t = \{(\tilde{y}_1^1, \ldots, \tilde{y}_t^1), (\tilde{y}_1^2, \ldots, \tilde{y}_t^2), \ldots, (\tilde{y}_1^k, \ldots, \tilde{y}_t^k)\}$$

At the next time step $t + 1$, for each hypothesis there will be $|V|$ possible y_{t+1} terms (V being the vocabulary), for a total of $k \cdot |V|$ possible hypotheses.

$$C_{t+1} = \bigcup_{i=1}^{k} \{(\tilde{y}_1^i, \ldots, \tilde{y}_t^i, v_1), \ldots, (\tilde{y}_1^i, \ldots, \tilde{y}_t^i, v_{|V|})\}$$

From these candidate sets, the decoding process keeps the k sequences with the highest probability. The process continues until each hypothesis reaches the special token representing the end of a sequence. We consider these k final sentences as candidate patches for the buggy code. Note that when $k = 1$, Beam Search decoding coincides with the greedy strategy.

Fig. 4 shows an example of the Beam Search decoding strategy with $k = 3$. Given the *abstract_b* code as input (top-left), the Beam Search starts by generating the top-3 candidates for the first

term (*i.e.,* public, void, private). At the next time step, the beam search expands each current hyphothesis and finds that the top-3 most likely are those following the node public. Therefore, the other two branches (*i.e.,* void, private) are pruned (*i.e.,* red nodes). The search continues till each hypothesis reaches the <eos> (End Of Sequence) symbol. Note that each hypothesis could reach the end at different time steps. This is a real example generated by our model, where one of the candidate patches is the actual fixed code (*i.e.,* green path).

2.3.4 Hyperparameter Search. For both models built on the BFP_{small} and BFP_{medium} dataset (*i.e.,* M_{small} and M_{medium}) we performed hyperparameter search by testing ten configurations of the encoder-decoder architecture. The configurations tested different combinations of RNN Cells (LSTM [31] and GRU [15]), 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 the configurations and settings are available in our online appendix [83].

2.3.5 Code Concretization. In this final phase, the abstracted code generated as output by the NMT model is concretized by mapping back all the identifiers and literal IDs to their actual values. The process simply replaces each ID found in the abstracted code to the real identifier/literal associated with the ID and saved in the mapping M, for each method pair. The code is automatically indented and additional code style rules can be enforced during this stage. While we do not deal with comments, they could be reintroduced in this stage as well.

3 EXPERIMENTAL DESIGN

The *goal* of this study is, as stated in the introduction, to empirically assess whether NMT can be used to learn fixes in the wild. The *context* consists of a dataset of bug fixes mined from Java open source projects hosted on GitHub (see Section Section 2).

The study aims at answering three research questions, described in the following.

3.1 RQ1: 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 rely on multiple internal and external datasets.

3.1.1 Internal Dataset. We use the two datasets BFP_{small} and BFP_{medium} to train and evaluate two NMT models M_{small} and M_{medium}. 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 a trained model (M_{small} or M_{medium}) and T be the test set of BFPs (BFP_{small} or BFP_{medium}), 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, performing inference with Beam Search Decoding for a given beam size k. The model will generate k different potential patches $P = \{abstract_p^1, \ldots, abstract_p^k\}$. We say that the model generated a successful fix for the code if there exists an $abstract_p^i \in P$ such that $abstract_p^i = abstract_f$. We report the raw count and percentage of successfully fixed BFPs in the test set, varying the beam size k from 1 (*i.e.,* a single patch is created by M) to 50 (*i.e.,* 50 patches are created) with incremental steps of 5.

Table 1. *CodRep* Datasets.

Datasets	M_{small}	M_{medium}
CodRep1	221	503
CodRep2	530	1130
CodRep3	665	1397
CodRep4	603	1124
CodRep5	1066	2119
CodRep	3027	6205

3.1.2 External Dataset. The external dataset comprises methods extracted from CodRep: a Machine Learning on Source Code Competition [14]. The goal of the competition is to predict where to insert a specific line into a source code file. The competition aims at being a common playground on which the machine learning and the software engineering research communities can interact [14], with potential usages in the field of automated program repair [13]. The dataset is composed of five collections of source code files taken from real commits in open-source projects, belonging to seven different studies in the literature focusing on bug-fixes and change history analysis [28, 46, 55, 71, 79, 95, 97]. On the *CodRep* dataset we perform the same steps described in Section 2.2, which involves extracting the changed methods, abstracting the pairs, and selecting only small and medium methods. Table 1 reports the number of unique (*i.e.,* no duplicates) method pairs in each of the five collections as well as the entire *CodRep* dataset (*i.e.,* last row in Table 1) where also inter-collections duplicates have been removed. The same models M_{small} and M_{medium}, which have been trained on the *BFP* dataset, are then evaluated on the entire *CodRep* dataset, without making any adjustments or adding more idioms. This helps in assessing the "portability" of our models when trained on a dataset D_i and tested on a different dataset D_j.

3.2 RQ2: What types of operations are performed by the models?

This RQ investigates the quality and type of operations performed by the fixes that our model generates. We perform the investigation by means of automated and manual analysis.

We first analyze the syntactic correctness of the patches for all beam widths. That is, we feed each potential patch $abstract_p^i$ to a Java lexer and parser in order to assess whether the patch is lexically and syntactically correct. We do not assess the compilability of the patches, since it would require us to download the exact, entire snapshot of each GitHub project. This would entail downloading thousands of different GitHub projects and attempting to compile them with the newly generated patch. There are also obstacles when dealing with different building systems.

Next, we focus on the BFPs that are successfully fixed by the models and analyze the types of AST operations performed during the fix. While these NMT models do not technically operate on the source code's AST, but rather on sequences of tokens, it is still worthwhile to understand the types of AST operations that such models can emulate. This analysis will provide an idea on the potential and/or limitations of such models. In detail, we extract the AST operations by selecting the action set A of the BFPs successfully fixed by the model. We identify the set M_A of unique AST actions performed by the model M in the successful fixes and compare them with the overall set O_A of unique AST operations contained within the entire test set of BFPs (*i.e.,* those that are needed to fix all the bugs in our test sets). With this information we can compute the percentage of AST actions in O_A that are learned and applied by M (*i.e.,* $|M_A|/|O_A|$). We also calculate the "theoretical bug coverage" ensured by M_A as the percentage of bugs in the test set that could be theoretically fixed by only using a subset of operations in M_A. This allows us to check whether the AST operations that are not "learned" by M (*i.e.,* $|O_A| \setminus |M_A|$) are used in many bug-fixing activities,
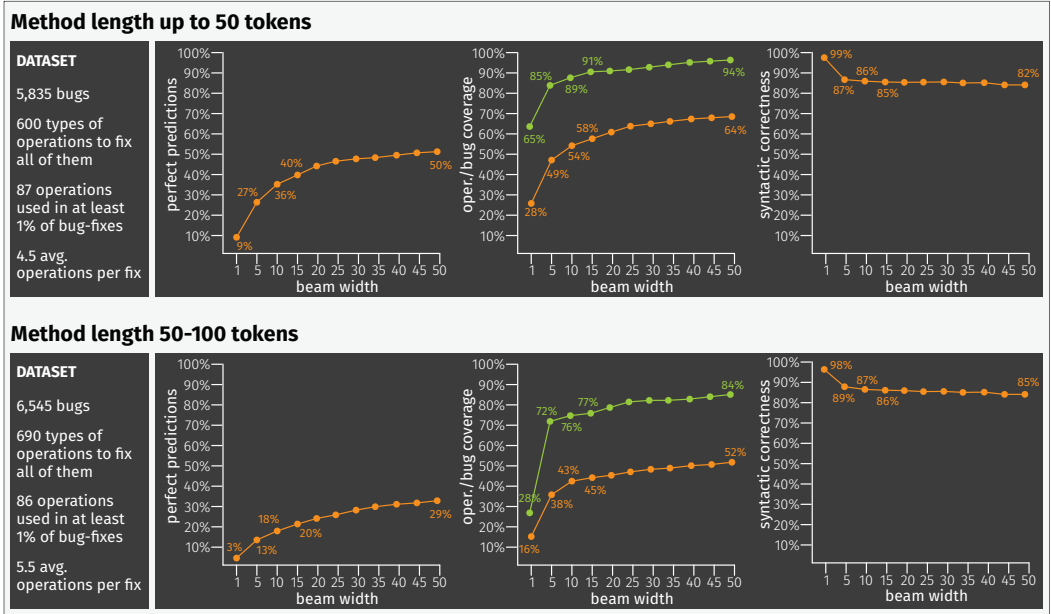
Fig. 5. Number of perfect prediction, operation (orange) bug (green) coverage, and syntactic correctness for varying beam width and for different method lengths.

thus representing an important loss for our model. A low theoretical bug coverage indicates that many bugs in test sets can not be fixed by only using the operations in M_A, while a high theoretical bug coverage points to the fact that the operations not learned by M are only sporadically used to fix bugs.

Finally, we discuss some interesting examples of the patches generated by NMT models.

3.3 RQ3: What is the training and inference time of the models?

In this RQ we evaluate the performance of the models in terms of execution time. Specifically, we analyze and discuss the time required to train the models, and the time needed to perform an inference once models have been deployed. For the latter, we report the total time of inference and compute the average time per patch generated for every beam width.

4 RESULTS

4.1 RQ1: 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, for both M_{small} and M_{medium}, was the one with 1-layer bi-directional Encoder, 2-layer Attention Decoder both with 256 units, embedding size of 512, and LSTM [31] RNN cells. We trained the M_{small} and M_{medium} models for 50k and 55k epochs, respectively.

4.1.1 Internal Dataset. Table 2 reports the number and percentage of BFPs correctly predicted by the models for different beam sizes. As expected, increasing the beam size and, therefore, generating more candidate patches, increases the percentages of BFPs for which the models can perfectly generate the corresponding fixed code starting from the buggy code input. The most surprising

Table 2. Models' Performances

Beam	BFP		CodRep	
	M_{small}	M_{medium}	M_{small}	M_{medium}
1	538 / 5835 (**9.22%**)	211 / 6545 (**3.22%**)	65 / 3027 (**2.14%**)	26 / 6545 (**0.41%**)
5	1595 / 5835 (**27.33%**)	859 / 6545 (**13.12%**)	311 / 3027 (**10.27%**)	207 / 6205 (**3.33%**)
10	2119 / 5835 (**36.31%**)	1166 / 6545 (**17.82%**)	450 / 3027 (**14.86%**)	361 / 6205 (**5.81%**)
15	2356 / 5835 (**40.37%**)	1326 / 6545 (**20.25%**)	539 / 3027 (**17.80%**)	451 / 6205 (**7.26%**)
20	2538 / 5835 (**43.49%**)	1451 / 6545 (**22.16%**)	614 / 3027 (**20.28%**)	524 / 6205 (**8.44%**)
25	2634 / 5835 (**45.14%**)	1558 / 6545 (**23.80%**)	657 / 3027 (**21.70%**)	574 / 6205 (**9.25%**)
30	2711 / 5835 (**46.46%**)	1660 / 6545 (**25.36%**)	701 / 3027 (**23.15%**)	599 / 6205 (**9.65%**)
35	2766 / 5835 (**47.40%**)	1720 / 6545 (**26.27%**)	733 / 3027 (**24.21%**)	644 / 6205 (**10.37%**)
40	2834 / 5835 (**48.56%**)	1777 / 6545 (**27.15%**)	761 / 3027 (**25.14%**)	679 / 6205 (**10.94%**)
45	2899 / 5835 (**49.68%**)	1830 / 6545 (**27.96%**)	784 / 3027 (**25.90%**)	709 / 6205 (**11.42%**)
50	2927 / 5835 (**50.16%**)	1869 / 6545 (**28.55%**)	807 / 3027 (**26.66%**)	749 / 6205 (**12.07%**)

results are those obtained with small beam sizes. The models can predict the fixed code of 9% and 3% of the BFPs with only one attempt. If we let the models generate 15 candidate patches, the percentage of perfect predictions bumps to 40% and 20% for small and medium methods, respectively. The number of BFPs patched steadily increases when more candidate patches are generated by the models (*i.e.,* bigger beam size), to reach a 50% and 28% of perfect predictions when 50 candidates patches are considered.

The leftmost graphs in Fig. 5 show the percentage of successful fixes as a function of the beam size. When setting the beam size to 50, M_{small} fixes 2,927 bugs (out of 5,835) in the same exact way they were fixed by developers. Likewise, M_{medium} fixes 1,869 bugs (out of 6,545). It is important to note that all BFPs in the test sets are unique and have never been seen before by the model during the training or validation steps. Moreover, there is no inherent upper bound to the beam width used during inference, therefore even larger beam widths could be set. All perfect predictions generated by the models at different beam sizes as well as experiments with even larger beam sizes are available in our online appendix [83].

The differences in performances between the M_{small} and M_{medium} could be explained by the fact that larger methods have potentially more faulty locations where a transformation of the code could be performed.

4.1.2 External Dataset. The last two columns of Table 2 report the results for the external dataset *CodRep*. The results show that the models M_{small} and M_{medium} – which have been trained on the test set of BFP_{small} and BFP_{medium} – are able to fix a high number of bugs belonging to a completely different and heterogeneous dataset. Similarly, increasing the beam size leads to improvements in the number of fixes, up to 26.66% and 12.07% for small and medium methods when 50 different potential patches are generated. Table 3 reports in details the fixes for each *CodRep* dataset (beam size 50). The overall percentages of fixes for *CodRep* dataset are slightly lower than those for *BFP*. This could be due to the fact that (i) the dataset is smaller, therefore, the model has fewer instances that can potentially fix; (ii) the dataset could contain different idioms which have not been considered (we did not fine-tune the idioms on this dataset). Overall, the results on the external dataset confirm the generalizability and potential of the NMT models to fix bugs on a different dataset.

Summary for RQ_1.

Using NMT, we trained a model on small BFPs, which can produce developer inspired fixes for 9.22% - 50.16% of bugs (dependent upon beam width). Likewise, a model trained on medium BFPs is capable of producing developer inspired fixes for 3.22% - 28.55% of bugs (dependent on beam

Table 3. *CodRep* Results

Datasets	M_{small}	M_{medium}
CodRep1	55 / 221 (**24.89**%)	48 / 503 (**9.54**%)
CodRep2	167 / 530 (**31.51**%)	136 / 1130 (**12.03**%)
CodRep3	180 / 665 (**27.07**%)	166 / 1397 (**11.88**%)
CodRep4	167 / 603 (**27.69**%)	157 / 1124 (**13.97**%)
CodRep5	288 / 1066 (**27.02**%)	244 / 2119 (**11.51**%)
CodRep	807 / 3027 (**26.66**%)	749 / 6205 (**12.07**%)

width). These results indicate that Neural Machine Translation is a viable approach for learning how to fix code. The results on the external dataset confirm the generalizability and potential of the NMT models.

4.2   RQ2: What types of operations are performed by the models?

Fig. 5 also shows the results of the two models (*i.e.,* M_{small} top, M_{medium} bottom) in terms of operations coverage, and syntactic correctness of the generated patches. Before discussing these results, it is important to comment on the dataset characteristics for small and medium BFPs. To fix the 5,835 small methods, developers adopted combinations of 600 different types of operations at the AST level (*e.g.,* Insert BinaryOperator at Conditional, Delete Catch at Try, *etc.*). Of these, only 87 have been used in more than 1% of bug-fixes, meaning that a vast majority of the AST operations have been rarely used to fix bugs (*e.g.,* in the case of the BFP_{small}, 513 types of AST operations have been used for the fixing of less than 58 bugs). Also, the average number of operations needed to fix a bug in the "small" dataset is 4.5. Similar observations can be done for BFP_{medium} (see Fig. 5).

4.2.1   Syntactic Correctness. We start by analyzing the syntactic correctness (rightmost graphs). We can notice that, when the models are asked to generate a single prediction (*i.e.,* the most likely one), the overall syntactic correctness of the predicted code is very high (99% and 98%). Clearly, the more candidate predictions the model is required to generate, the more likely is that it introduces syntactic errors during the transformation of the code. We observe this phenomenon in the graph with a decreasing syntactic correctness, reaching 82% and 85% when 50 variants of patches are generated. The slightly better syntactic correctness achieved by the M_{medium} model could be explained by the fact that, in larger methods, there are more potential points of transformation where syntactically correct variants can be generated, with respect to smaller methods. While we do not measure the compilability rate of the generated patches, it is worth to note that the perfect predictions generated by the models correspond to the code that was actually committed to repositories by developers. For such reasons, we could reasonably expect those predicted patches to be compilable.

4.2.2   AST Operations. The center graphs in Fig. 5 show the operation coverage (orange line) and theoretical bug coverage (green line) when varying the beam size. When only one candidate patch is generated, the models M_{small} and M_{medium} cover 28% and 16% of the total unique operations in the entire test sets, which include 600 and 690 operations, respectively. An increase of the beam size to 5 and 10 leads to a dramatic surge in the coverage of various operations in the test set. These results show that allowing the models to generate more candidate patches not only leads to more fixes, but also to a larger variety of bug fixing operations being performed. The operation coverage keeps increasing with larger beam widths.

Table 4. Perfect Prediction Top-10 Operations

	Delete
Frequency	Operation
1774	Delete Invocation at Block
1295	Delete TypeAccess at ThisAccess
1034	Delete FieldRead at Invocation
878	Delete VariableRead at Invocation
850	Delete TypeAccess at Invocation
789	Delete Literal at Invocation
622	Delete TypeAccess at FieldRead
602	Delete ThisAccess at FieldRead
570	Delete ThisAccess at Invocation
532	Delete Literal at BinaryOperator

	Insert
Frequency	Operation
338	Insert TypeAccess at ThisAccess
186	Insert Literal at BinaryOperator
185	Insert Block at Method
156	Insert ThisAccess at FieldRead
155	Insert BinaryOperator at If
150	Insert ThisAccess at Invocation
147	Insert If at Block
136	Insert VariableRead at Invocation
123	Insert Return at Block
114	Insert Block at If

	Move
Frequency	Operation
277	Move Invocation from Block to CtInvocationImpl
129	Move Block from If to CtBlockImpl
97	Move Return from Block to CtReturnImpl
76	Move Invocation from BinaryOperator to CtInvocationImpl
71	Move Invocation from Invocation to CtInvocationImpl
59	Move Parameter from Method to CtParameterImpl
59	Move BinaryOperator from BinaryOperator to CtBinaryOperatorImpl
57	Move Invocation from LocalVariable to CtInvocationImpl
57	Move Block from Method to CtBlockImpl
40	Move Method from Class to CtMethodImpl

	Update
Frequency	Operation
314	Update Invocation at Block
303	Update Method at Class
122	Update Invocation at Invocation
114	Update Literal at Invocation
77	Update BinaryOperator at If
75	Update Invocation at Return
48	Update Literal at BinaryOperator
45	Update BinaryOperator at BinaryOperator
38	Update Invocation at LocalVariable
33	Update ThisAccess at FieldRead

We observe a similar trend for the theoretical bug coverage, with large improvements in the early beam widths, and a steady increase afterwards. It is also worth to carefully speculate on the theoretical bug coverage percentages. As a matter of fact, the results suggest that – with combinations of the AST actions learned and successfully emulated by the models in perfect fixes – the models could theoretically cover 94% and 84% of the bug fixes in the test set. This means that the AST operations that the models failed to learn are quite rare, and only used in a small subset of the real bug-fixing activities used for training.

Fig. 6. Examples of successfully-generated patches.

Table 4 reports the top-10 operations for each category successfully emulated by the models when generating the perfect predictions. The complete list of all the AST operations as extracted by GumTree Diff is available in our online appendix [83]

4.2.3 Qualitative Examples. Fig. 6 shows some interesting examples of patches generated by the model. For space limitations, we focus on interesting fixes distilled from the set of perfect predictions generated by the model M_{small}. The examples are shown in abstracted code (with idioms), as they are fed and generated by the models. The actual source code can be generated by mapping back all the IDs to the real values stored in the mapping M. Fig. 6 also groups the examples based on the "type of fix" implemented, showing the ability of the model in learning different fixing patterns, also in the context of the same group. For example, we show that not all fixes dealing with if

conditions are identical. All examples are perfect predictions meaning that the model changed the buggy method to reflect exactly how the developer changed the method in the wild.

Our first group of examples (1-6) concern buggy methods that were missing and if or that benefited from its addition. Thus, the added condition either helped to prevent errors during execution or ensured the expected outcome from the executed code. Example 1 shows an in-line if condition added to the fixed method to check whether the getValue method is being called on a null object and returns 0 if it is. If the object is not null, then the original getValue method is called on the object and that value is returned. This fix ensures that the get method is not called on a null object. Likewise, example 2 inserts a similar check but targeting the length of a variable rather than a getter method. The in-line if checks ensures the variable is not null, if it is then the method returns 0, otherwise, the method returns the result of VAR_1.length(). This patch is interesting because the string VAR_1 is set to null before the length is calculated. This could be an example where the developer was performing error checking and wanted to ensure that the string was in fact nulled out. Given the context, we would expect the method to always return 0. However, the original method calls the length() method on a null string, which would return an error. Although this patch is "correct", the model may have learned a sub-optimal coding practice from the developer's patch. The model's generated patch is unsurprising as the training data included many in-line if checks as a potential fix. Most likely, the model identified this pattern and applied a similar fix in this scenario. Examples 3, 4 and 5 all insert similar if-checks that are not in-line ifs. Examples 3 and 5 both add an if condition handling cases in which the invoked method returns null, while example 4's if condition checks the size of a variable before operating on it.

The last example in this group (*i.e.,* number 6) is different from the others, since the if condition is more complex and makes use of the boolean operator not (!). Here the fix is preventing the method from adding a duplicated value to VAR_2. If the value VAR_1 is already present in VAR_2, then the method will not add VAR_1 again. It is important to note that although these examples all add an if check as the fix, they are all unique and tailored to the method's context. The model was able to learn the correct changes needed for the specific method that would mimic a developer's changes.

The second group of fixes addresses issues related to the cast of a specific variable. The original method in example 7 would throw an error if executed because the method signature calls for a return value of type float, but the method returns a value of type int. The model recognized this error and casted the return value of type float. Additionally, the fix also changes the mechanism by which the value is extracted from values (see Fig. 6). This not only changes the type of value returned by also the mechanism by which it is returned.

The next group of examples pertain to the implementation or structure of the code, leading to incorrect execution. Example 8 switches the statements' order of execution, without applying any other change. This swap could be needed due to the first statement changing the state of the system (*e.g.,* the value of VAR_1) which would then cause the VAR_1.METHOD_2(VAR_2) invocation to have a different outcome. Our model is capable of finding such errors in order execution and provide an adequate fix. Example 9 is similar in that the structure of the code is incorrect. Here the switch statement is missing a default case in the buggy method. Thus, the buggy method will execute the switch and, if no case condition will be met, the code outside the switch statement will be run. The fix adds a default case to the switch statement to handle cases in which no case condition is met. This fix does not change the outcome of the code since the code executed outside the case statement (buggy version) and inside the default statement (fixed version) is exactly the same (see Fig. 6). However, it improves the readability of the code, making it adhering to the Java coding convention suggesting that switch statements should have a default case, which occurs when no other case in the switch has been met.

Our fourth category of examples are changes where the model fixes `try-catch` statements. We report one representative example (number 10). This fix changes the scope of the `try` block to also include in it the `for` loop, that was instead containing the `try` block in the buggy method.

The fifth group of fixes we found addresses incorrect `else` statements. In example 11 we see that the `else if` statement is removed from the buggy method. This change is seen as a bug fix since the buggy method only defines its behavior when VAR_1 < VAR_2 or VAR_1 > VAR_2. It has no behavior defined when VAR_1 == VAR_2, which could lead to unexpected errors. The model fixes this by replacing the `else if` statement with an `else`, covering all possible relations between VAR_1 and VAR_2.

The sixth group of fixes aims at replacing incorrect method calls. As seen in example 12, the method call METHOD_1 is replaced with `min`. The example demonstrates the power of idioms. Indeed, without this idiom, we would discard this fix since we would be unable to generate a name for the unseen method `min` in the fix and would name it METHOD_3. Since METHOD_3 would not be seen in our mapping *M*, we would have to synthesize the new methods name when translating the abstracted code back into source code. Having `min` as an idiom allows us to avoid the synthesis and still learn the fix. Although this patch replaces METHOD_1 with `min`, which seems very specific, it is logical that the method is being replaced with a mathematical method. The power of our RNN is that context is taken into account and the context of this method demonstrates a mathematical operation. When the model observes mathematical methods it may be inclined to generate a patch using variables and method calls seen in similar methods. Therefore, the `min` function makes sense as a patch since it compares two numbers and is a mathematical function, which fits the context of the overall method well. Example 13 shows instead the removal of unnecessary/harmful method calls. Here the model removes in the fixed method the invocation to `intValue()` on VAR_2. This method is used to return a numeric value, represented by an object, as an `int`. In this situation, textttVAR_2 is a Java integer object and `intValue()` would return an `int` type. The fix removes this method call which compares an object to `int`, making `equals` comparing VAR_1 to the integer type VAR_2.

Finally, the last group of fixes involves the changing, addition or removal of logic or boolean operators. Although they changes themselves do not appear massive, they have major implications on the source code behavior. For instance, example 14 adds a negation boolean operator to the `if` condition. This completely changes the functionality of the fixed method since now it will only execute the `if` block when ! VAR_2 == true. Example 15 performs a fix along the same line, changing an operator in the `if` condition from logical or (||) to a logical and (&&). This means that both conditions must be met in order to run the code within the `if` block. Since the buggy method allowed only one condition to be met, it is possible that this led to undesired results for the developers. Example 16 changes a <= to a < operator. It is worth noting that this operator change takes place within the scope of a `while` loop, thus reducing by one the times that the code in the `while` loop is executed.

The reported qualitative examples show the potential of NMT models to generate meaningful correct patches, by learning from real bug-fixes wrote by developers, which allows the model to avoid problems arising with existing program repair techniques. Indeed, a previous work by Qi *et al.* [67] found that existing techniques achieve repair by overfitting on the test cases, or by simply deleting pieces of functionality. The models produced many other interesting patches, which are not discussed here due to space limitations. Our online appendix [83] contains many more examples of bug-fixes using different operations, considering methods with different lengths, and using a variety of beam widths.

Besides success cases, we also qualitatively discuss code snippets for which our approach failed in fixing the bug. Each of the discussed examples is shown in Figure 7 in three lines: the first

Fig. 7. Examples of unfixed bugs.

represent the buggy code to fix, the second is the fix generated by our approach, and the third is the correct fix implemented by the developer.

In the first example, the model deleted the last `if` statement from the buggy code. However, the developer fix aimed at renaming the method, removing VAR_1 from the method parameters and deleting the `if`-block at the end of the method body. These three changes were not completely captured by the model. In fact, the only fix that was implemented was the removal of the if-block at the end of the method body. Most likely, the model failed to completely fix this bug due to the drastic changes to the original method and a lack of training data to properly identify this pattern of changes. From a practical standpoint, the renaming of the method is a refactoring operation rather than a bug fix, and the model's fix would have the same functionality as the developer fix. Although the model's fix is not perfect, the functionality is correct.

In the second example, the model's fix changes the return statement from an integer of unknown value, to the idiom 0. However, the developer's fix is much more extensive as it assigns to value the integer cast of VAR_2. After this integer cast of VAR_2 has been performed, value is returned. In this example, the developer has introduced a completely new statement and a variable as a bug fix. The model struggles with synthesizing new statements unless the pattern is repeatedly seen in the training data. It is likely that this pattern of changes was not seen frequently during training and, therefore, the model performed the fix incorrectly.

The third example demonstrates the model's inability to recognize the need for an `if`-block in the beginning of the method body. The model attempted to fix the bug by changing the type of TYPE_1 . info (this , VAR_1). The developer's fix shows that the type of that variable is correct. However, an `if`-check was missing to determine whether the out variable was null. Without this check, methods could be called on a null object, which would lead to an error. The model has previously inserted these `if`-checks in other examples; however, in this particular case, the model failed to recognize the potential for a null object.

The fourth example shows the model incorrectly changing the return value. The developer's performed fix on this method was to add the Boolean not operator: ! to VAR_2. Our model changed the and operator && to a . equals ( ) method call. The model's fix is the opposite of the developer's fix: the developer's method will return true whereas the model's method would return false, and *vice versa*. One explanation for the model's behavior could be the frequent case of changing == to . equals ( ), which we see often in the training data. Although the && symbol is not the same as ==, the same symbol twice in a row pattern may cause the model to replace that symbol with . equals ( ).

Finally, the fifth example shows that the model struggles to find various, drastic changes that were implemented by the developer. The model's "fix" in this situation is to remove the type from the assertThat method call. However, the developer has fixed this method by replacing the METHOD_1.get(STRING_1) with METHOD_1 . isEmpty ( ) and then replacing the STRING_2 argument in TYPE_2 . METHOD_2 (STRING_2) with the idiom true. The model fails to recognize both changes and therefore would be ineffective in finding a bug fix for the original method. The developer's method completely changes the computation and the purpose from the original method, which may be why the model is not capable to perform a fix.

Summary for RQ$_2$. The models exhibit a very high syntactic correctness of the generated patches ranging between 99% and 82%. Moreover, while the models are able to learn on how to apply a subset of the AST operation types exploited by developers to fix all bugs in the test set, the learned operations are the most representative ones, allowing to, theoretically, fix a large percentage of bugs.

Fig. 8. Inference Time (M_{medium}).

4.3 RQ3: What is the training and inference time of the models?

The training of the models M_{small} and M_{medium} took six and 15 hours respectively, running on a server with three consumer-level GPUs. Overall, this is an acceptable one-time cost that allows building a cross-project bug-fixing model in a reasonable amount of time. Fig. 8 shows the average inference time per patch (orange line) and per bug (green line) for the M_{medium} model with increasingly large beam size. While the average time per bug rises with larger beam sizes (*i.e.,* more patches generated for the same bug) from a minimum of only 0.006s ($k = 1$) to a maximum of 0.226s ($k = 30$), the average time per patch generated stays well below 0.030s. Overall, the model is able to generate 50 candidate patches for a bug in less than a second. The inference times for M_{small} are even lower. The complete timing results, raw values, and total number of seconds are available in our online appendix [83].

Summary for RQ$_3$. After training for less than 15 hours, the models are able to generate 50 candidate patches for a single bug in less than a second.

5 THREATS TO VALIDITY

Construct validity threats concern the relationship between theory and observation, and are mainly related to likely sources of imprecision in our analyses. To have enough training data, we mined bug-fixes in GitHub repositories rather than using curated bug-fix datasets such as Defects4j [34] or IntroClass[44], 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 validity threats concern factors internal to our study that could influence our results. It is possible that the performance of our models depends on the hyperparameter configuration. We explain in Section 2.3.4 how hyperparameter search has been performed.

External validity threat concern the generalizability of our findings. 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, or the execution of the test cases for checking a patch's suitability. This is 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.

Finally, we only focused on small- and medium-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 RELATED WORK

This section describes related work on (i) automated program repair techniques and, specifically, their underlying redundancy assumption, and (ii) the use of machine translation to support software engineering tasks.

6.1 Program Repair and the Redundancy Assumption

Automated program repair involves the transformation of an unacceptable behavior of a program execution into an acceptable one according to a specification [54]. Behavioral repair techniques in particular change the behavior of a program under repair by changing its source or binary code [54]. These techniques [20, 45, 73] rely on a critical assumption, the *redundancy assumption*, that claims large programs contain the seeds of their own repair. This assumption has been examined by at least two independent empirical studies, showing that a significant proportion of commits originates from previously-existing code [7, 51].

Martinez *et al.* [51] empirically examined the assumption that certain bugs can be fixed by copying and rearranging existing code. They validated the redundancy assumption by defining a concept of *software temporal redundancy*. A commit is temporally redundant if it is a rearrangement of code in previous commits. They measured redundancy at two levels of granularity: line- and token-level. At line-level granularity, they found that most of the temporal redundancy is localized in the same file. At token-level granularity, their results imply that many repairs never need to invent a new token.

Barr *et al.* [7] examined a history of 15,723 commits to determine the extent to which the commits can be reconstructed from existing code. The grafts they found were mostly single lines, *i.e.,* micro-clones, and they proposed that micro-clones are useful since they are the atoms of code construction [7]. Their findings align with Martinez *et al.* [51] in that changes to a codebase contain fragments that already exist in the code base at the time of the change.

Repair approaches based on the redundancy assumption are called redundancy-based repair techniques, since they leverage redundancy and repetition in source code [7, 12, 18, 30, 51, 60, 65, 90]. For example, GenProg [20, 21, 45] searches for statement-level modifications to make to an abstract syntax tree. The approach by Arcuri and Yao [5] co-evolves programs and test cases using a model

similar to the predatory-prey one. Weimer *et al.* [86] perform program repair using a deterministic search, reducing the search space with program equivalence analysis. Le *et al.* [42] use the content of previous patches to reward patches that can have a likely better acceptability for developers, and therefore avoid over-fitting patches to test cases. A complementary set of repair techniques leverage program analysis and program synthesis to repair programs by constructing code with particular properties [36, 41, 52, 53, 61, 64, 78, 93].

Some approaches perform automated program repair by searching the fix among manually-written patterns, as in the case of PAR [37], or through a SMT (satisfiability modulo theories)-based semantic code search over a large body of existing code, as for SearchRepair [36]. Instead, Prophet [48] is a learning-based approach that uses explicitly designed code features to rank candidate repairs. Other approaches train on correct solutions (from student programs) to specific programming tasks and try to learn task-specific repair strategies [8, 66]. This goal has been achieved successfully in contexts such as in massively open online courses (MOOC), where the programs are generally small and synthetic [27].

Finally, one of the first approaches able to learn code transformation templates from human-written patches is Genesis, by Long *et al.* [47]. Genesis can learn patches related to three types of defects, specifically related to null pointers, out of bounds and class casting. Long *et al.* show how Genesis is able to find more patches than previously-proposed approaches like PAR [37]. Also, the approach proposed by Long *et al.* is capable to infer new tokens from existing patches. Our approach is different from the work of Long *et al.*, as we employ NMT whereas they use integer programming. Also, differently from Genesis, in our work we would like to experiment with the applicability of NMT to learn and then apply generic (and not defect-specific) fixes from a large body of source code changes.

The goal of our empirical investigation was to determine whether NMT could be used to bring the "redundancy assumption", but also the heuristics used by program repair approaches using code search, at a next level. Such a next level would be the capability to automatically learn patches from large software corpora.

As mentioned in the introduction, this work represent an extension of our previous work in which we proposed the general idea of learning bug-fixes using NMT [82]. While our previous paper mainly presented the idea and assessed its overall feasibility, this paper reports an extensive evaluation, in which we also (i) generate multiple candidate patches via beam search; (ii) analyze the types of AST operations performed in the fixes as well as the syntactic correctness; (iii) qualitatively analyze the kinds of fix operations the learned models are able to perform, and (vi) assess the timing performance of the approach when learning the models and when recommending the fix.

6.2 Machine Translation in Software Engineering

Modern machine translation systems generally use data-driven methods to translate text or speech from one language to another. Machine translation systems are trained on translated texts, or "parallel corpora", for particular text types [38] in both natural languages and formal languages such as programming languages. Manually migrating software projects from one language to another is a time-consuming and error-prone task [96]. Nguyen *et al.* [57–59] used *statistical* machine translation for method-to-method migration from Java to C#, translating small token sequences at a time. Recently, NMT systems superseded traditional statistical approaches as the state-of-the-art in translation. One advantage neural systems have over purely statistical systems is they can measure fluency at a higher level of granularity, *e.g.,* sentence-level granularity, rather than being constrained to phrases. However, NMT systems are indeed data-hungry systems, and this problem has been an issue for software engineering applications where there are not a lot of parallel corpora. DeepAM [26] uses deep learning to automatically mine application programming

interface mappings from a source code corpus without parallel text. Our work is intended to study the feasibility of using NMT to learn bug-fixes from real-world changes.

7 FUTURE WORK

Our initial goal is to continue to improve and analyze the model we have generated. One task we plan to begin with is to analyze the trade off between our training set percentage, model quality and training time. We plan to use a tiered approach to determine how many training instances are needed to achieve an optimal performance. Therefore, we will train the model with 10% of the training set, then increase this percentage to 20%, 30%, etc. all the way up to 90%. This analysis will include a discussion about diminishing returns as well as the timing required to train the model. We currently don't implement this techinique in our work due to the extensive amount of time needed for the study. Training deep neural networks can take many days and is heavily reliant on the computational power and size of the dataset. As our dataset is fairly large, training on 10% of the training set all the way up to 90% will take a substantial amount of time. Then after the model is trained, we would still need to perform the analysis to determine the quality of the model. Given that this is a very expensive analysis, we save this task for future work.

As extensive future work, we want to focus on two major limitations of the approach. The first is to address a different level of granularity as opposed to function level. Currently, our approach only learns patches at a function level with less than 100 tokens. We would like to increase this size and maybe even change the granularity to class or package level. This change in granularity would increase the context for certain bug fixes and should make the approach more meaningful to developers. It may even be possible to generate bug fixes that span across multiple files, requiring a change to multiple methods.

Our second task is similar to the first, but we want to focus on learning smaller changes within the context of a larger method. We want to focus on a segmentation technique, which would allow us to keep the context meaningful but still allow the model to change tokens. Our idea is to further abstract the source code so that only meaningful statements are recognized. This technique can be difficult since further abstraction leads to less meaningful context. However, we believe that walking the line between less meaningful context and not overwhelming the model, is the key to success. We do not want to lose fixes we have found by changing types and variables within the code, therefore, we plan to combine our tool with state of the art static analysis tools. These latter should be able to find meaningful fixes pertaining to types and variables, which frees up our deep learning model to focus on more complex fixes. Thus, we plan to severely abstract any statement or code fragment that can be changed with a static analysis tool and feed the remaining parts of the large method into our deep learning RNN model. This methodology still maintains meaningful context while allowing the model to focus on more complex parts of the method, which static analysis may not be useful for.

8 CONCLUSION

We presented an extensive empirical investigation into the applicability of Neuro-Machine Translation (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-fix pairs (BFPs). Then, we set up, trained, and tuned NMT models to *translate* buggy code into fixed code. Our empirical analysis aimed at assessing the feasibility of the NMT technique applied to the bug-fixing problem, the types and quality of the predicted patches, as well as the training and inference time of the models.

We found the models to be able to fix a large number of unique bug-fixes, ranging between 9-50% of small BFPs (up to 2,927 unique fixed bugs) and 3-28% of medium BFPs (up to 1,869 unique

fixed bugs) in our test set, depending on the amount of candidate patches we require the model to generate. The models generate syntactically correct patches in more than 82% of the cases. The model M_{small} is able to emulate between 28-64% of the Abstract Syntax Tree operations performed during fixes, while M_{medium} achieves between 16-52% of the coverage. Finally, the running time analysis shows that these models are capable of generating tens of candidate patches in a split of a second.

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

9 ACKNOWLEDGMENT

REFERENCES

[1] Alali, A., Kagdi, H. H., and Maletic, J. I. 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* (2008), pp. 182–191.

[2] Allamanis, M. The adverse effects of code duplication in machine learning models of code.

[3] Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2015), ESEC/FSE 2015, ACM, pp. 38–49.

[4] Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., and Guéhéneuc, Y. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research, October 27-30, 2008, Richmond Hill, Ontario, Canada* (2008), p. 23.

[5] Arcuri, A., and Yao, X. A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2008, June 1-6, 2008, Hong Kong, China* (2008), pp. 162–168.

[6] Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. *CoRR abs/1409.0473* (2014).

[7] Barr, E. T., Brun, Y., Devanbu, P., Harman, M., and Sarro, F. The plastic surgery hypothesis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014), FSE 2014, ACM, pp. 306–317.

[8] Bhatia, S., and Singh, R. Automated correction for syntax errors in programming assignments using recurrent neural networks. *CoRR abs/1603.06129* (2016).

[9] Boulanger-Lewandowski, N., Bengio, Y., and Vincent, P. Audio chord recognition with recurrent neural networks. In *ISMIR* (2013), Citeseer, pp. 335–340.

[10] Britz, D., Goldie, A., Luong, M., and Le, Q. V. Massive exploration of neural machine translation architectures. *CoRR abs/1703.03906* (2017).

[11] Brown, D. B., Vaughn, M., Liblit, B., and Reps, T. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2017), ESEC/FSE 2017, ACM, pp. 511–522.

[12] Carzaniga, A., Gorla, A., Mattavelli, A., Perino, N., and Pezzè, M. Automatic recovery from runtime failures. In *Proceedings of the 2013 International Conference on Software Engineering* (Piscataway, NJ, USA, 2013), ICSE '13, IEEE Press, pp. 782–791.

[13] Chen, Z., and Monperrus, M. CodRep. https://github.com/KTH/CodRep-competition, 2018.

[14] Chen, Z., and Monperrus, M. The codrep machine learning on source code competition. Tech. Rep. 1807.03200, arXiv, 2018.

[15] Cho, K., van Merrienboer, B., Gülçehre, Ç., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR abs/1406.1078* (2014).

[16] Falleri, J., Morandat, F., Blanc, X., Martinez, M., and Monperrus, M. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014* (2014), pp. 313–324.

[17] FISCHER, M., PINZGER, M., AND GALL, H. C. 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* (2003), p. 23.

[18] GABEL, M., AND SU, Z. A study of the uniqueness of source code. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2010), FSE '10, ACM, pp. 147–156.

[19] GITHUB. GitHub Compare API. https://developer.github.com/v3/repos/commits/#compare-two-commits, 2010.

[20] GOUES, C. L., DEWEY-VOGT, M., FORREST, S., AND WEIMER, W. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. ICSE'12.

[21] GOUES, C. L., WEIMER, W., AND FORREST, S. Representations and operators for improving evolutionary software repair. GECCO'12.

[22] GRAVES, A. Sequence transduction with recurrent neural networks. *CoRR abs/1211.3711* (2012).

[23] GRIGORIK, I. GitHub Archive. https://www.githubarchive.org, 2012.

[24] GU, X., ZHANG, H., AND KIM, S. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 3, 2018* (2018).

[25] GU, X., ZHANG, H., ZHANG, D., AND KIM, S. 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* (2016), pp. 631–642.

[26] GU, X., ZHANG, H., ZHANG, D., AND KIM, S. DeepAM: migrate APIs with multi-modal sequence to sequence learning. *CoRR abs/1704.07734* (2017).

[27] GUPTA, R., KANADE, A., AND SHEVADE, S. K. Deep reinforcement learning for programming language correction. *CoRR abs/1801.10467* (2018).

[28] HATA, H., MIZUNO, O., AND KIKUNO, T. Bug prediction based on fine-grained module histories. *Proceedings - International Conference on Software Engineering* (06 2012), 200–210.

[29] HERZIG, K., JUST, S., AND ZELLER, A. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013* (2013), pp. 392–401.

[30] HINDLE, A., BARR, E. T., SU, Z., GABEL, M., AND DEVANBU, P. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering* (Piscataway, NJ, USA, 2012), ICSE '12, IEEE Press, pp. 837–847.

[31] HOCHREITER, S., AND SCHMIDHUBER, J. Long short-term memory. *Neural Comput. 9*, 8 (Nov. 1997), 1735–1780.

[32] JIN, G., SONG, L., ZHANG, W., LU, S., AND LIBLIT, B. Automated atomicity-violation fixing. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 389–400.

[33] JORGENSEN, M., AND SHEPPERD, M. A systematic review of software development cost estimation studies. *IEEE Trans. Softw. Eng. 33*, 1 (Jan. 2007), 33–53.

[34] JUST, R., JALALI, D., AND ERNST, M. D. 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* (New York, NY, USA, 2014), ISSTA 2014, ACM, pp. 437–440.

[35] KALCHBRENNER, N., AND BLUNSOM, P. Recurrent continuous translation models. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing* (Seattle, Washington, USA, October 2013), Association for Computational Linguistics, pp. 1700–1709.

[36] KE, Y., STOLEE, K., LE GOUES, C., AND BRUN, Y. Repairing programs with semantic code search. ASE'15.

[37] KIM, D., NAM, J., SONG, J., AND KIM, S. Automatic patch generation learned from human-written patches. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013* (2013), pp. 802–811.

[38] KOEHN, P. *Statistical Machine Translation.* 2010.

[39] KOLASSA, C., RIEHLE, D., AND SALIM, M. A. A model of the commit size distribution of open source. In *SOFSEM 2013: Theory and Practice of Computer Science* (Berlin, Heidelberg, 2013), P. van Emde Boas, F. C. A. Groen, G. F. Italiano, J. Nawrocki, and H. Sack, Eds., Springer Berlin Heidelberg, pp. 52–66.

[40] LAM, A. N., NGUYEN, A. T., NGUYEN, H. A., AND NGUYEN, T. N. 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* (2017), pp. 218–229.

[41] LE, X., CHU, D., LO, D., LE GOUES, C., AND VISSER, W. S3: Syntax- and semantic-guided repair synthesis via programming by examples. FSE'17.

[42] LE, X. D., LO, D., AND LE GOUES, C. 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* (2016), pp. 213–224.

[43] LE GOUES, C., DEWEY-VOGT, M., FORREST, S., AND WEIMER, W. 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* (2012), pp. 3–13.

[44] LE GOUES, C., HOLTSCHULTE, N., SMITH, E., BRUN, Y., DEVANBU, P., FORREST, S., AND WEIMER, W. The manybugs and introclass benchmarks for automated repair of C programs. *TSE 41*, 12 (2015), 1236–1256.

[45] Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng. 38*, 1 (2012), 54–72.

[46] Li, D., Li, L., Kim, D., Bissyandé, T. F., Lo, D., and Traon, Y. L. Watch out for this commit! A study of influential software changes. *CoRR abs/1606.03266* (2016).

[47] Long, F., Amidon, P., and Rinard, M. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2017), ESEC/FSE 2017, ACM, pp. 727–739.

[48] Long, F., and Rinard, M. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2016), POPL '16, ACM, pp. 298–312.

[49] Luong, M., Pham, H., and Manning, C. D. Effective approaches to attention-based neural machine translation. *CoRR abs/1508.04025* (2015).

[50] Martinez, M., Durieux, T., Sommerard, R., Xuan, J., and Monperrus, M. Automatic repair of real bugs in Java: a large-scale experiment on the Defects4J dataset. *Empirical Software Engineering 22*, 4 (2017), 1936–1964.

[51] Martinez, M., Weimer, W., and Monperrus, M. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Companion Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE Companion 2014, ACM, pp. 492–495.

[52] Mechtaev, S., Jooyong, Y., and Roychoudhury, A. Angelix: Scalable multiline program patch synthesis via symbolic analysis. ICSE'16.

[53] Mechtaev, S., Jooyong, Y., and Roychoudhury, A. DirectFix: Looking for simple program repairs. ICSE'15.

[54] Monperrus, M. Automatic software repair: A bibliography. *ACM Comput. Surv. 51*, 1 (Jan. 2018), 17:1–17:24.

[55] Monperrus, M., and Martinez, M. CVS-Vintage: A Dataset of 14 CVS Repositories of Java Software. working paper or preprint, Dec. 2012.

[56] Moran, K., Bernal-Cárdenas, C., Curcio, M., Bonett, R., and Poshyvanyk, D. Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Transactions on Software Engineering* (2018).

[57] Nguyen, A. T., Nguyen, H. A., Nguyen, T. T., and Nguyen, T. N. Statistical learning approach for mining API usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (New York, NY, USA, 2014), ASE '14, ACM, pp. 457–468.

[58] Nguyen, A. T., Nguyen, T. T., and Nguyen, T. N. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2013), ESEC/FSE 2013, ACM, pp. 651–654.

[59] Nguyen, A. T., Nguyen, T. T., and Nguyen, T. N. Migrating code with statistical machine translation. In *Companion Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE Companion 2014, ACM, pp. 544–547.

[60] Nguyen, H. A., Nguyen, A. T., Nguyen, T. T., Nguyen, T. N., and Rajan, H. A study of repetitiveness of code changes in software evolution. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (Piscataway, NJ, USA, 2013), ASE'13, IEEE Press, pp. 180–190.

[61] Nguyen, H. D. T., Qi, D., Roychoudhury, A., and Chandra, S. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering* (Piscataway, NJ, USA, 2013), ICSE '13, IEEE Press, pp. 772–781.

[62] Parr, T. *The Definitive ANTLR 4 Reference*, 2nd ed. Pragmatic Bookshelf, 2013.

[63] Parr, T., and Fisher, K. Ll(*): The foundation of the ANTLR parser generator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 425–436.

[64] Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W.-F., Zibin, Y., Ernst, M. D., and Rinard, M. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 87–102.

[65] Pierret, D., and Poshyvanyk, D. An empirical exploration of regularities in open-source software lexicons. In *The 17th IEEE International Conference on Program Comprehension, ICPC 2009, Vancouver, British Columbia, Canada, May 17-19, 2009* (2009), pp. 228–232.

[66] Pu, Y., Narasimhan, K., Solar-Lezama, A., and Barzilay, R. Sk_p: A neural program corrector for MOOCs. SPLASH Companion 2016.

[67] Qi, Z., Long, F., Achour, S., and Rinard, M. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. ISSTA'15.

[68] Raychev, V., Vechev, M., and Yahav, E. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14,

ACM, pp. 419–428.

[69] Roy, C. K., and Cordy, J. R. NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008* (2008), pp. 172–181.

[70] Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K., and Lopes, C. V. SourcererCC: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering* (New York, NY, USA, 2016), ICSE '16, ACM, pp. 1157–1168.

[71] Scholtes, I., Mavrodiev, P., and Schweitzer, F. From aristotle to ringelmann: a large-scale analysis of team productivity and coordination in open source software projects. *Empirical Software Engineering 21*, 2 (Apr 2016), 642–683.

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

[73] Sidiroglou-Douskos, S., Lahtinen, E., Long, F., and Rinard, M. Automatic error elimination by horizontal code transfer across multiple applications. *SIGPLAN Not. 50*, 6 (June 2015), 43–54.

[74] Smith, E. K., Barr, E. T., Le Goues, C., and Brun, Y. 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* (New York, NY, USA, 2015), ESEC/FSE 2015, ACM, pp. 532–543.

[75] Sobreira, V., Durieux, T., Delfim, F. M., Monperrus, M., and de Almeida Maia, M. 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* (2018), pp. 130–140.

[76] Soto, M., and Le Goues, C. Using a probabilistic model to predict bug fixes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2018), IEEE, pp. 221–231.

[77] Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. *CoRR abs/1409.3215* (2014).

[78] Tian, Y., and Ray, B. Automatically diagnosing and repairing error handling bugs in C. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2017), ESEC/FSE 2017, ACM, pp. 752–762.

[79] Tufano, M., Bavota, G., Poshyvanyk, D., Di Penta, M., Oliveto, R., and De Lucia, A. An empirical study on developer-related factors characterizing fix-inducing commits. *Journal of Software: Evolution and Process 29*, 1, e1797. e1797 JSME-15-0185.R2.

[80] Tufano, M., Pantiuchina, J., Watson, C., Bavota, G., and Poshyvanyk, D. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering* (2019), ICSE 19, ACM.

[81] Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., and Poshyvanyk, D. Deep learning similarities from different representations of source code. In *Proceedings of the 15th International Conference on Mining Software Repositories* (New York, NY, USA, 2018), MSR '18, ACM, pp. 542–553.

[82] Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., and Poshyvanyk, D. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (New York, NY, USA, 2018), ASE 2018, ACM, pp. 832–837.

[83] Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., and Poshyvanyk, D. Online Appendix. https://sites.google.com/view/learning-fixes, 2018.

[84] van Bruggen, D. JavaParser. https://javaparser.org/about.html, 2014.

[85] Wang, S., Liu, T., and Tan, L. 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* (2016), pp. 297–308.

[86] Weimer, W., Fry, Z. P., and Forrest, S. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (Piscataway, NJ, USA, 2013), ASE'13, IEEE Press, pp. 356–366.

[87] Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings* (2009), pp. 364–374.

[88] Weiss, A., Guha, A., and Brun, Y. Tortoise: Interactive system configuration repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Piscataway, NJ, USA, 2017), ASE 2017, IEEE Press, pp. 625–636.

[89] Weiss, C., Premraj, R., Zimmermann, T., and Zeller, A. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories* (Washington, DC, USA, 2007), MSR '07, IEEE Computer Society, pp. 1–.

[90] White, M., Tufano, M., Martinez, M., Monperrus, M., and Poshyvanyk, D. Sorting and transforming program repair ingredients via deep learning code similarities. In *2019 IEEE 26th International Conference on Software Analysis,*

Evolution and Reengineering (SANER) (2019), IEEE, p. to appear.

[91] WHITE, M., TUFANO, M., VENDOME, C., AND POSHYVANYK, D. 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* (2016), pp. 87–98.

[92] WHITE, M., VENDOME, C., LINARES-VÁSQUEZ, M., AND POSHYVANYK, D. Toward deep learning software repositories. MSR'15.

[93] XUAN, J., MARTÍNEZ, M., DEMARCO, F., CLÉMENT, M., LAMELAS, S., DURIEUX, T., LE BERRE, D., AND MONPERRUS, M. Nopol: Automatic repair of conditional statement bugs in Java programs. *IEEE Trans. Software Engineering 43*, 1 (2016), 34–55.

[94] YANG, J., ZHIKHARTSEV, A., LIU, Y., AND TAN, L. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2017), ESEC/FSE 2017, ACM, pp. 831–841.

[95] ZHONG, H., AND SU, Z. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Piscataway, NJ, USA, 2015), ICSE '15, IEEE Press, pp. 913–923.

[96] ZHONG, H., THUMMALAPENTA, S., XIE, T., ZHANG, L., AND WANG, Q. Mining API mapping for language migration. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1* (New York, NY, USA, 2010), ICSE '10, ACM, pp. 195–204.

[97] ZHOU, J., ZHANG, H., AND LO, D. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering* (Piscataway, NJ, USA, 2012), ICSE '12, IEEE Press, pp. 14–24.