Learning Code Transformations via Neural Machine Translation

Michele Tufano

Avellino, Italy

Bachelor in Computer Science, University of Salerno, 2012
Master in Computer Science, University of Salerno, 2014

A Dissertation presented to the Graduate Faculty
of The College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

College of William & Mary
April 2019

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Michele Tufano

Approved by the Committee, April 2019

Committee Chair
Associate Professor Denys Poshyvanyk, Computer Science
College of William & Mary

Chair and Associate Professor Robert Michael Lewis, Computer Science
College of William & Mary

Associate Professor Gang Zhou, Computer Science
College of William & Mary

Assistant Professor Xu Liu, Computer Science
College of William & Mary

Assistant Professor Gabriele Bavota, Computer Science
Università della Svizzera italiana (USI)

ABSTRACT

Source code evolves – *inevitably* – to remain useful, secure, correct, readable, and efficient. Developers perform software evolution and maintenance activities by *transforming* existing source code via corrective, adaptive, perfective, and preventive changes. These code changes are usually managed and stored by a variety of tools and infrastructures such as version control, issue trackers, and code review systems. Software evolution and maintenance researchers have been mining these code archives in order to distill useful insights regarding the nature of developers' activities. One of the long-lasting goals of software engineering research is to better support and *automate* different types of code changes performed by developers. In this thesis we depart from classic manually crafted rule- or heuristic-based approaches, and propose a novel technique to *learn* code transformations by leveraging the vast amount of publicly available code changes performed by developers. We rely on Deep Learning, and in particular on Neural Machine Translation (NMT), to train models able to learn code change patterns and apply them to novel, unseen, source code.

First, we tackle the problem of generating source code mutants for Mutation Testing. In contrast to classic approaches, which rely on handcrafted mutation operators, we propose to automatically learn how to mutate source code by observing real faults. We mine millions of bug fixing commits from GitHub, and then process and abstract this source code. This data is used to train and evaluate an NMT model to translate fixed code into buggy code (*i.e.*, the mutated code). In the second project, we rely on the same dataset of bug-fixes to learn code transformations for the purpose of Automated Program Repair (APR). This represents one of the most challenging research problem in Software Engineering, whose goal is to automatically fix bugs without developers' intervention. We train a model to translate buggy code into fixed code (*i.e.*, learning patches) and, in conjunction with beam search, generate many different potential patches for a given buggy method. In our empirical investigation we found that such a model is able to fix thousands of unique buggy methods in the wild. Finally, in our third project we push our novel technique to the limits by enlarging the scope to consider not only bug-fixing activities, but any type of meaningful code changes performed by developers. We focus on accepted and merged code changes that have undergone a Pull Request (PR) process. We quantitatively and qualitatively investigate the code transformations learned by the model to build a taxonomy. The taxonomy shows that NMT can replicate a wide variety of meaningful code changes, especially refactorings and bug-fixing activities.

In this dissertation we illustrate and evaluate the proposed techniques, which represent a significant departure from earlier approaches in the literature. The promising results corroborate the potential applicability of learning techniques, such as NMT, to a variety of Software Engineering tasks.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

This Dissertation would not have been possible without the advice and support of several people. First, I would like to thank my advisor Dr. Denys Poshyvanyk. He has been an exceptional mentor and friend to me during the ups and downs of my Ph.D. journey. Denys supported me with confidence and trust during the inevitable difficulties and challenging times of the Ph.D. studies. Equally important, he always made sure I was keeping my feet on the ground during the successful times, helping me to remain focused on my objectives and work. Starting the Ph.D. has been a life-changing decision that I would have never taken without the suggestion and support of Dr. Gabriele Bavota. Thank you, Gabriele, for everything. My interest and passion for Software Engineering started when I attended Prof. Andrea De Lucia's classes. Thanks Andrea. I would also like to thank my committee members, Robert, Gang, and Xu, for their valuable feedback on this Dissertation.

This dissertation is the result of a joint effort and dedication of several collaborators. Thanks Cody for your incredible work and commitment to our projects. Thanks Massimiliano, you've been a figure of expertise and competence I've always relied upon. Thanks Marty, you've been a teacher and inspiration for me. In my early years of Ph.D. I've also had the opportunity to collaborate with an outstanding researcher and friend, Fabio Palomba. Thanks Fabio for the achievements we share.

I would like to thank the entire SEMERU group. Other than those already mentioned, I would like to thank Carlos, Kevin, and Chris for their friendship and great research collaboration. Thanks David, Boyang, and Qi for making the lab 101C such a great environment for working and laughing together. My gratitude goes also to my lifelong friends *VIPPT* who always encouraged me. I miss you guys!

Finally, and most importantly, I would like to thank my family. You supported me in this adventure even if it meant being far away from you, on the other side of the world. Everything I've ever achieved is thanks to your guide and love.

To my Mother and Father.

LIST OF TABLES

LIST OF FIGURES

Learning Code Transformations via Neural Machine Translation

Chapter 1

Introduction

1.1 Overview

Software and its source code evolves continuously in order to remain useful, secure, and correct for users as well as readable and maintainable for developers and contributors. Developers perform a myriad of different types of changes on the source code during software evolution and maintenance activities. These operations are usually classified [124] according to the following categories :

Corrective changes that aim at correcting or fixing discovered problems (*i.e.*, bugs) in the system;

Adaptive changes which are intended to keep a software product useful in a changing environment (*e.g.*, requirements, standards);

Perfective changes that are performed in order to improve performance or maintainability of the code;

Preventive changes that aim to detect and correct latent faults in the software product before they become effective faults.

We refer to all these types of changes as *code transformations*. These transformations are usually managed and stored by a variety of tools and infrastructures such as version

control (*e.g.*, git [19], svn [30]), issue trackers (*e.g.*, Bugzilla [14], Jira [24]), and code review systems (*e.g.*, Gerrit [18]). The advent and continuous success of open source software projects has made enormous amount of data publicly available for practitioners and researchers to analyze. Certainly, many researchers have mined and analyzed code archives in order to distill useful insights on the nature of code changes and developers' activities. For example, some have studied the nature of commits [91, 206], others analyzed the changes and diffs performed by developers [99, 159], while others have focused on particular types of changes such as bug-fixes [121, 100] and refactorings [70].

One of the long-lasting goals of Software Engineering research is to better support and automate different types of code changes performed by developers, such as bug-fixes, refactoring, testing *etc.* However, the foundation of state of the art techniques are generally based upon manually crafted rules or heuristics defined by human experts. This not only means limited automation, but also intrinsic bias introduced by the human experts who crafted these rules and mechanisms.

In this thesis we propose to adopt Deep Learning techniques and models to automatically learn from thousands of code changes. In particular, we aim to learn what the important code features that represents the source code are, without manually specifying what we – as researchers or developers – believe is important. We propose to learn change patterns by observing thousands of real world code transformations performed by real developers. With this knowledge, we aim to automatically apply code transformations on novel, unseen code, with the goal of automating code transformations that would be otherwise performed by developers.

Given a software change that transforms the *code before* the change into the *code after* the change, we represent this as a Neural Machine Translation problem, where the code before is the *original text*, the code after is a *translated text*, and the change itself is the translation process. This idea is depicted in Figure 1.1.

Neural Machine Translation is an end-to-end deep learning approach for automated translation. In this thesis we apply Neural Machine Translation to learn code transforma-

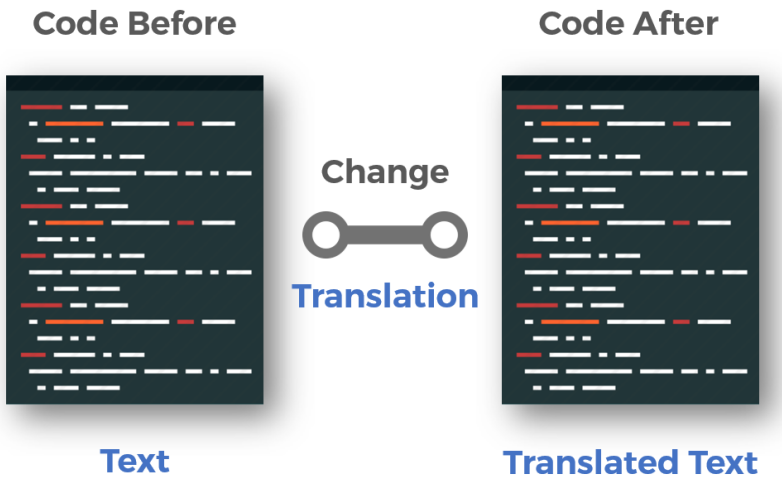

Figure 1.1: Code Transformations as Neural Machine Translation problem tions in the context of three Software Engineering tasks: Mutation Testing, Automated Program Repair, and Learning Code Changes.

1.2 Motivation – Learning for Automation

The main goal of the work described in this thesis is to devise a set of techniques that are able to learn code transformations from real world data, with the final objective of *automating* a variety of software engineering tasks.

This work is motivated primarily by the fact that the literature's landscape is dominated by techniques relying on manually handcrafted rules or heuristics.

1.2.1 Mutation Testing

Mutation Testing is a strategy used to evaluate the quality of a test suite, which involves the creation of modified versions of the tested code (*i.e.*, mutants) and observing how well the existing test suite is able to detect the inserted mutants. Classic approaches rely on well-defined mutation operators to generate the code variants, by applying these operators in all, or random, possible locations in the code. These operators are intended to mimic usual program mistakes and are defined by human experts which usually derive

them through observation of real bugs, past experience or knowledge of the programming constructs and errors. For example, a state of the art mutation tool, Pit [32], offers 22 mutation operators manipulating different programming constructs, such as: conditional statements (*i.e.*, Conditionals Boundary Mutator, Negate Conditionals Mutator), mathematical operations (*i.e.*, Math Mutator, Increments Mutator), return statements (*i.e.*, Empty returns Mutator, False Returns Mutator), *etc.* While these mutation operators range across many types of potential bugs, they are still limited and somewhat trivial. Existing mutation approaches generate too many and too simple mutants, which sometimes are not even compilable or worth to execute. These approaches are also limited in their applicability to automation, in the sense that the types and locations of the mutation operators need to be selected. Finally, and most importantly, there is no evidence that these mutation operators are actually similar to real bugs and follow similar changes and distribution.

We propose a novel approach to automatically learn mutants from faults in real programs, by observing that a buggy code can arguably represent the perfect mutant for the corresponding fixed code.

1.2.2 Automated Program Repair

Automated Program Repair (APR) represents one of the most challenging research problem in Software Engineering, whose goal is to automatically fix bugs without developers' intervention. Similarly to Mutation Testing, existing APR techniques are mostly based on hard-coded rules and operators – defined by human experts – used to generate potential patches for a buggy code. For example, GenProg [121] works at statement-level by inserting, removing or replacing a statement taken from other parts of the same program. MutRepair [131] attempts to generate patches by applying mutation operators on suspicious if-condition statements. SemFix [144] instead uses symbolic execution and constraint solving to synthesize a patch by replacing only the right-hand side of assignments or branch predicates. These approaches have several limitations, rooted in the fact that

they are based on handcrafted rules with limited scope (*i.e.*, single statements, limited number of mutation operators, specific parts of expressions). Moreover, Smith *et al.* [168] showed that these approaches tend to overfit on test cases, by generating incorrect patches that pass the test cases, mostly by deleting functionalities.

Conversely, we aim to automatically learn how to fig bugs by observing thousands of real bug-fixes performed by developers in the wild.

1.2.3 Code Changes

As we discussed previously, there are many different types of changes, outside of those focusing on testing and bug-fixing. Adaptive, perfective, and preventive changes can vary widely based on the context and their final objective. In this realm, there exist plenty of tools and infrastructures that attempt to automate or support developers in performing particular types of changes. Tools for Refactoring [49], API Evolution [200], Code Smell removal [173], Performance, and Optimization depend on hard-coded rules that apply in specific situations.

We aim to train NMT models to learn a wide variety of types of changes which apply in different contexts by training our models on software changes that have been reviewed and accepted by developers during code review process.

1.3 Contributions & Outline

In this thesis we devise a technique that enables to learn code transformations via Neural Machine Translation. We mine a large dataset of code transformations, such as bug-fixes and reviewed code changes, from publicly available repositories. These changes are then analyzed with a fine-grained AST differencing tool and appropriately clustered together. We devise a technique called *Code Abstraction* which is intended to transform the original source code into a representations that is suitable for NMT and at the same time preserve syntactic and semantic information of the code. Finally, we train and evaluate NMT

models that are able to predict and replicate a large variety of bug-fixes, mutants, and other changes. The results show that the models can emulate a large set of AST operations and generate syntactically correct code.

In **Chapter 2** we provide background related to Neural Machine Translation and details on how we apply this technique on source code.

In **Chapter 3** we develop a novel technique to learn how to generate mutants from bug-fixes. This technique relies on NMT to learn how to translate fixed code into the buggy version of the same code, by observing millions of real world examples of bug-fixes performed by developers in open source repositories. After we mined millions of bug-fixes from GitHub, we process this data and perform *code abstraction*, a procedure we developed that makes NMT feasible on source code. The bug-fixes are then clustered together based on their AST operations and different models are trained for each cluster. The content of this chapter is based primarily on the paper [183].

In **Chapter 4** we devise an Automated Program Repair (APR) technique that learns how to fix bugs by observing real world examples of developers' bug-fixes, rather than relying on manually crafted rules and heuristics. The technique is based on a NMT model which works in conjunction with beam search to generate many different translations (*i.e.*, patches) of a given buggy method. The content of this chapter is based primarily on the paper [181, 182].

In **Chapter 5** we study the potential of NMT models to learn a wide variety of code changes. To this goal, we mine accepted and merged code changes that have undergone a Pull Request (PR) process, from Gerrit, a code review system. We then train NMT models and investigate – both quantitatively and qualitatively – the number and types of code changes successfully learned by the model and devise a taxonomy. The content of this chapter is based primarily on the paper [178].

In addition to the contributions outlined in this dissertation, the author has worked on a wide array of research topics in software engineering over the course of his career as a doctoral student including: (i) code smells and anti-patterns both in production

[177, 174, 148] and test code [175]; (ii) deep learning code similarities [196, 180, 195]; (iii) software compilation and building [176, 179]; (iv) Android testing [186, 138] and performance [125, 51].

Chapter 2

Background

2.1 Neural Machine Translation

Neural Machine Translation (NMT) is an end-to-end deep learning approach for automated translation. It outperforms phrase-based systems without the need of creating handcrafted features such as lexical or grammatical rules. It has been used not only in language translations – such as Google Translate [23] – but also in text summarization [162], question-answering tasks [47], and conversational models [187]. Basically any problem in which one wants to learn a mapping between an input sequence of terms and a corresponding output sequence of terms could be potentially mapped as a Neural Machine Translation problem.

NMT was introduced by Kalchbrenner *et al.* [106], Sutskever *et al.* [171], and Cho *et al.* [59] who defined Recurrent Neural Networks (RNN) models for machine translation. The most famous architecture and what we used in our study was introduced by Bahdanau *et al.* [46] in 2015.

At the very high level, NMT models are comprised of an Encoder and a Decoder, both are Recurrent Neural Networks that are trained jointly. An attention mechanism helps aligning the input tokens to the output tokens in order to facilitate the translation. The Encoder reads the input sentence and generates a sequence of hidden states. These hidden

states are then used by the Decoder to generate a sequence of output words, representing the translation of the input sentence.

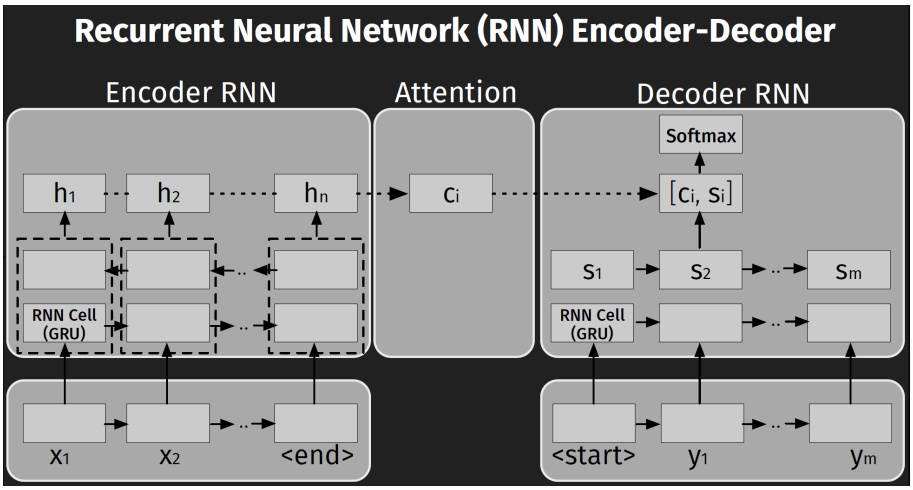

Figure 2.1: Neural Machine Translation model

2.1.1 RNN Encoder-Decoder

The models trained and evaluated in this thesis are based on an RNN Encoder-Decoder architecture, commonly adopted in NMT [106, 171, 59]. This model consists of two major components: an RNN Encoder, which *encodes* a sequence of terms $\boldsymbol{x}$ into a vector representation, and an RNN Decoder, which *decodes* the representation into another sequence of terms $\boldsymbol{y}$. The model learns a conditional distribution over a (output) sequence conditioned on another (input) sequence of terms: $P(y_1, .., y_m | x_1, .., x_n)$, where n and m may differ. In our case, we would like to learn the code transformation $code_{before} \rightarrow code_{after}$, therefore given an input sequence $\mathbf{x} = code_{before} = (x_1, .., x_n)$ and a target sequence $\mathbf{y} = code_{after} = (y_1, .., y_m)$, the model is trained to learn the conditional distribution: $P(code_{after} | code_{before}) = P(y_1, .., y_m | x_1, .., x_n)$, where x_i and y_j are abstracted source tokens: Java keywords, separators, IDs, and idioms. Fig. 2.1 shows the architecture of the Encoder-Decoder model with attention mechanism [46, 128, 54]. 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

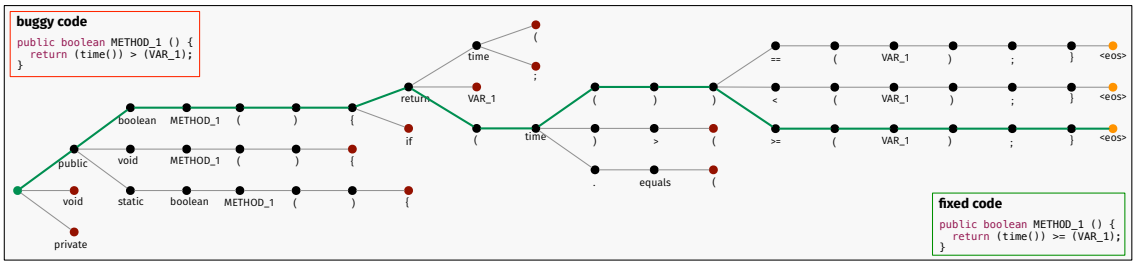

Figure 2.2: Beam Search Visualization.

on a bi-directional RNN Encoder [46], which is formed by a backward and a forward RNN, which are able to create representations taking into account both past and future inputs [54]. That is, each state h_i represents the concatenation (dashed box in Fig. 2.1) 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.

2.1.2 Generating Multiple Translations via Beam Search

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 $code_{before}$ as input, the trained model will generate only one possible sequence of predicted $code_{after}$. Conversely, we would like to generate multiple potential translations (*i.e.*, candidates for $code_{after}$) for a given $code_{before}$. To this aim, we employ a different decoding strategy called Beam Search and used in previous applications of deep learning [158]. 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.

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

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. 2.2 shows an example of the Beam Search decoding strategy with $k = 3$. Given the $code_{before}$ that represents a buggy code 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 $code_{after}$ is the actual fixed code (*i.e.*, green path).

2.1.3 Hyperparameter Search

Table 2.1 shows the 10 different configurations of hyperparameters we employed when tuning the models. Each row represents a configuration with the following fields:

- **ID**: Configuration number;
- **Embedding**: the size of the embedding vector used to represent each token in the sentence;
- **Encoder and Decoder**: Encoder and Decoder settings:
  - *Layers*: number of layers;
  - *Units*: number of units (*i.e.*, neurons);
- **Cell**: Type of RNN cell used (*e.g.*, LSTM or GRU);

Table 2.1: Hyperparameter Configurations

ID	Embedding	Encoder		Decoder		Cell
		Layers	Units	Layers	Units	
1	256	1	256	2	256	GRU
2	256	1	256	2	256	LSTM
3	256	2	256	4	256	GRU
4	256	2	256	4	256	LSTM
5	256	2	512	4	512	GRU
6	256	2	512	4	512	LSTM
7	512	2	512	4	512	GRU
8	512	2	512	4	512	LSTM
9	512	1	256	2	256	GRU
10	512	1	256	2	256	LSTM

It is worth noting that, while there can be many more possible configurations, our goal was to experiment with a diverse set of values for each parameter in a reasonable time. More fine tuning will be needed in future work. In defining the grid of hyperparameter values, we relied on the suggestions available in the literature [54], in particular, we define

the Decoder to be deeper (*i.e.*, number of layers) than the Encoder as experimental studies suggest.

We found that the configuration yielding the best results is the #10: 512 embedding size, 1-layer Encoder, 2-layers Decoder both with 256 neurons and LSTM cells. It is interesting to note that this represents the *shallowest* model we tested, but with the *biggest* dimensionality of embedding size. We believe this is due to two major factors: (i) the relatively small number of training instances w.r.t. the model parameters makes larger models effectively not being able to generalize on new instances; (ii) a larger dimensionality will result in a more expressive representation for sentences, and in turn, better performances.

In recent experiments we tested an even smaller model (based on configuration #10) with only 128 neurons for each layer with the goal of minimizing the model and obtain better generalization. We observed a slight drop in performances ($\sim$5%) but faster training time. Further experiments shall be performed in order to tune the parameters of the model and find the sweet spot between size and performances.

2.1.4 Overfitting

> *All standard neural network architectures such as the fully connected multi-layer perceptron are prone to overfitting [74]: While the network seems to get better and better, i.e., the error on the training set decreases, at some point during training it actually begins to get worse again, i.e., the error on unseen examples increases.*
>
> — Lutz Prechelt, Early Stopping - But When? [155]

This phenomenon is depicted in Figure 2.3, where the error (*i.e.*, loss function) on the training and validation set is observed throughout the training iterations. The error on the training set steadily decreases during the training iterations, on the other hand, the error on unseen examples belonging to the validation set starts to get worse at some point during the training. In other words, the model stops generalizing and, instead, starts learning the

statistical noise in the training dataset. This *overfitting* of the training dataset will result in an increase in generalization error, making the model less useful at making predictions on new data points.

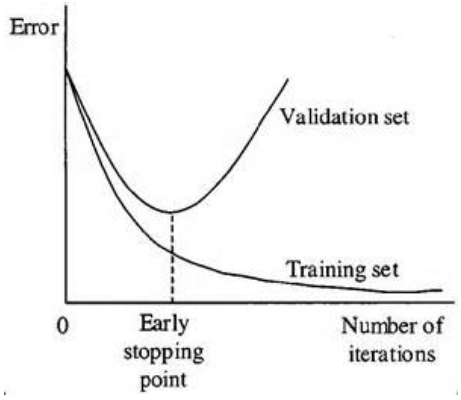

Figure 2.3: Overfitting

Intuitively, we would like to stop the training right before the model starts to overfit on the training instances (*i.e.*, the early stopping point in Figure 2.3).

> *This strategy is known as early stopping. It is probably the most commonly used form of regularization in deep learning. Its popularity is due both to its effectiveness and its simplicity.*
>
> — Goodfellow *et al.*, Deep Learning [77]

In this work, we employ early stopping to avoid overfitting. In particular, we let the model train for a maximum of 60k iterations and select the model's checkpoint with the minimum error on the *validation* set – not the *training* set – and finally testing this selected model on the unseen data in the *test* set.

Chapter 3

Learning How to Mutate Source Code from Bug-Fixes

3.1 Introduction

Mutation testing is a program analysis technique aimed at injecting artificial faults into the program's source code or bytecode [89, 64] to simulate defects. Mutants (*i.e.*, versions of the program with an artificial defect) can guide the design of a test suite, *i.e.*, test cases are written or automatically generated by tools such as Evosuite [71, 72] until a given percentage of mutants has been "killed". Also, mutation testing can be used to assess the effectiveness of an existing test suite when the latter is already available [139, 53].

A number of studies have been dedicated to understand the interconnection between mutants and real faults [41, 42, 63, 104, 127, 166, 105, 57]. Daran and Thévenod-Fosse [63] and Andrews *et al.* [41, 42] indicated that mutants, if carefully selected, can provide a good indication of a test suite's ability to detect real faults. However, they can underestimate a test suite's fault detection capability [41, 42]. Also, as pointed out by Just *et al.* [105], there is a need to improve mutant taxonomies in order to make them more representative of real faults. In summary, previous work suggests that mutants can be representative of real faults if they properly reflect the types and distributions of faults that programs

exhibit. Taxonomies of mutants have been devised by taking typical bugs into account [145, 110]. Furthermore, some authors have tried to devise specific mutants for certain domains [98, 188, 65, 146, 186]. A recent work by Brown *et al.* [55] leveraged bug-fixes to extract syntactic-mutation patterns from the diffs of patches. The authors mined 7.5k types of mutation operators that can be applied to generate mutants.

However, devising specific mutant taxonomies not only requires a substantial manual effort, but also fails to sufficiently cope with the limitations of mutation testing pointed out by previous work. For example, different projects or modules may exhibit diverse distributions of bugs and types, helping to explain why defect prediction approaches do not work out-of-the-box when applied cross-project [205]. Instead of manually devising mutant taxonomies, we propose to automatically learn mutants from existing bug fixes. Such a strategy is likely to be effective for several reasons. First, a massive number of bug-fixing commits are available in public repositories. In our exploration, we found around 10M bug-fix related commits on GitHub just in the last six years. Second, a buggy code fragment arguably represents the perfect mutant for the fixed code because: (i) the buggy version is a mutation of the fixed code; (ii) such a mutation already exposed a buggy behavior; (iii) the buggy code does not represent a trivial mutant; (iv) the test suite did not detect the bug in the buggy version. Third, advanced machine learning techniques such as deep learning have been successfully applied to capture code characteristics and effectively support several SE tasks tasks [189, 196, 115, 83, 81, 158, 92, 38, 39, 35].

Stemming from such considerations, and being inspired from the work of Brown *et al.* [55], we propose an approach for automatically learning mutants from actual bug fixes. After having mined bug-fixing commits from software repositories, we extract change operations using an AST-based differencing tool and abstract them. Then, to enable learning of specific mutants, we cluster similar changes together. Finally, we learn from the changes using a Recurrent Neural Network (RNN) Encoder-Decoder architecture [106, 171, 59]. When applied to unseen code, the learned model decides in which location and what changes should be performed. Besides being able to learn mutants from an existing

source code corpus, and differently from Brown *et al.* [55], our approach is also able to determine where and how to mutate source code, as well as to introduce new literals and identifiers in the mutated code.

We evaluate our approach on $787k$ bug-fixing commits with the aim of investigating (i) how similar the learned mutants are as compared to real bugs; (ii) how specialized models (obtained by clustering changes) can be used to generate specific sets of mutants; and (iii) from a qualitative point of view, what operators were the models able to learn. The results indicate that our approach is able to generate mutants that perfectly correspond to the original buggy code in 9% to 45% of cases (depending on the model). Most of the generated mutants are syntactically correct (more than 98%), and the specialized models are able to inject different types of mutants.

This chapter provides the following contributions:

- A novel approach for learning how to mutate source code from bug-fixes. To the best of our knowledge, this is the first attempt to automatically learn and generate mutants.
- Empirical evidence that our models are able to learn diverse mutation operators that are closely related to real bugs.
- We release the data and code to enable replication [43].

3.2 Approach

We start by mining bug-fixing commits from thousands of GitHub repositories (Sec. 3.2.1). From the bug-fixes, we extract method-level pairs of *buggy* and corresponding *fixed* code that we call *transformation pairs* (TPs) (Sec. 3.2.2.1). TPs represent the examples we use to learn how to mutate code from bug-fixes (*fixed* $\rightarrow$ *buggy*). We rely on GumTree [67] to extract a 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 TPs (Sec. 3.2.2.2)

into a representation that is more suitable for learning. The output of this phase is the set of abstracted TPs and their corresponding mapping M which allows to reconstruct the original source code. Next, we generate different datasets of TPs (Sec. 3.2.2.4 and 3.2.2.5). Finally, for each set of TPs we use an encoder-decoder model to learn how to transform a *fixed* piece of code into the corresponding *buggy* version (Sec. 3.2.3).

3.2.1 Bug-Fixes Mining

We downloaded the GitHub Archive [80] containing every public GitHub event between March 2011 and October 2017. Then, we used the Google BigQuery APIs to identify commits related to bug-fixes. We selected all the commits having a message containing the patterns: ("fix" or "solve") and ("bug" or "issue" or "problem" or "error"). We identified 10,056,052 bug-fixing commits for which we extracted the commit ID (SHA), the project repository, and the commit message.

Since we are aware that not all commit messages matching our pattern are necessarily related to corrective maintenance [44, 93], we assessed the precision of the regular expression used to identify bug-fixing commits. Two authors independently analyzed a statistically significant sample (95% confidence level ±5% confidence interval, for a total size of 384) of identified commits to judge whether the commits were actually referring to bug-fixing activities. Next, the authors met to resolve a few disagreements in the evaluation (only 13 cases). The evaluation results, available in our appendix [43], reported a true positive rate of 97%. The commits classified as false positives mainly referred to partial and incomplete fixes.

After collecting the bug-fixing commits, for each commit we extracted the source code pre- and post- bug-fixing (*i.e.*, buggy and fixed code) by using the GitHub Compare API [75]. During this process, we discarded files that were created in the bug-fixing commit, since there is no buggy version to learn from, as the mutant would be the deletion of the entire source code file. In this phase, we also discarded commits that had touched more than five Java files, since we aim to learn from bug-fixes focusing on only a few files and

not spread across the system, and as found in previous work [94], large commits are more likely to represent tangled changes, *i.e.*, commits dealing with different tasks. Also, we excluded commits related to repositories written in languages different than Java, since we aim at learning mutation operators for Java code. After these filtering steps, we extracted the pre- and post-code from ∼787k (787,178) bug-fixing commits.

3.2.2 Transformation Pairs Analysis

A TP 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 TPs as examples when training our RNN. The idea is to train the model to learn the transformation from the fixed code component (m_f) to the buggy code (m_b), in order to generate mutants that are similar to real bugs.

3.2.2.1 Extraction

Given a bug-fix *bf*, we extracted the buggy files (f_b) and the corresponding fixed (f_f) files. For each pair (f_b, f_f), we ran AST differencing between the ASTs of f_b and f_f using GumTree Spoon AST Diff [67], to compute the sequence of AST edit actions that transforms f_b into f_f.

Instead of computing the AST differencing between the entire buggy and fixed files, we separate the code into method-level pieces that will constitute our TPs. 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 mapped method m_if (from the fixed file f_f). Next, for each pair of mapped methods, we extract a sequence of edit actions using the GumTree algorithm. We then consider only those method pairs for which there is at least one edit action (*i.e.*, we disregard methods unmodified during the fix). Therefore, the output of this phase is a list of $TPs = \{tp_1, \ldots, tp_k\}$, where each TP is a triplet $tp = \{m_b, m_f, A\}$, 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 do not

consider any methods that have been newly created or completely deleted within the fixed file, since we cannot learn mutation operations from them. Also, TPs do not capture changes performed outside methods (*e.g.*, class name).

The rationale behind the choice of method-level TPs is manyfold. First, methods represent a reasonable target for mutation, since they are more likely to implement a single task. Second, methods provide enough meaningful context for learning mutations, such as variables, parameters, and method calls used in the method. Smaller snippets of code lack the necessary context. Third, file- or class-level granularity could be too large to learn patterns of transformation. Finally, considering arbitrarily long snippets of code, such as hunks in diffs, could make the learning more difficult given the variability in size and context [113, 33]. Note that we consider each TP as an independent fix, meaning that multiple methods modified in the same bug fixing activity are considered independently from one other. In total, we extracted ∼2.3M TPs.

3.2.2.2 Abstraction

The major problem in dealing with raw source code in TPs is the extremely large vocabulary created by the multitude of identifiers and literals used in the code of the ∼2M mined projects. This large vocabulary would hinder our goal of learning transformations of code as a neural machine translation task. Therefore, we abstract the code and generate an expressive yet vocabulary-limited representation. We use a combination of a Java lexer and parser to represent each buggy and fixed method within a TP, as a stream of tokens. First, the lexer (based on ANTLR [152, 150]) reads the raw code tokenizing it into a stream of tokens. The tokenized stream is then fed into a Java parser [185], which discerns the role of each identifier (*i.e.*, whether it represents a variable, method, or type name) and the type of literals.

Each TP is abstracted in isolation. Given a TP $tp = \{m_b, m_f, A\}$, we first consider the source code of m_f. 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 then 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 ($abstract_f$). Then, we consider the source code of m_b. The Java lexer produces a stream of tokens, which is then fed to the parser. The parser continues to use map M for m_b. The parser generates new IDs only for novel identifiers/literals, not already contained in M, meaning, they exist in m_b but not in m_f. Then, it replaces all the identifiers/literals with the corresponding IDs, generating the abstracted method ($abstract_b$). The abstracted TP is now the following 4-tuple $tpa = \{abstract_b, abstract_f, A, M\}$, where M is the ID mapping for that particular TP. The process continues considering the next TP, generating a new mapping M. Note that we first analyze the fixed code m_f and then the corresponding buggy code m_b of a TP since this is the direction of the learning process (from m_b to m_f).

The assignment of IDs to identifiers and literals occurs in a sequential and positional fashion. Thus, the first method name found will receive the ID `METHOD_1`, likewise the second method name will receive ID `METHOD_2`. This process continues for all method and variable names (`VAR_X`) and literals (`STRING_X`, `INT_X`, `FLOAT_X`). Figure 3.1 shows an example of the TP's abstracted code. It is worth noting that IDs are shared between the two versions of the methods and new IDs are generated only for newly found identifiers/literals. The abstracted code allows to substantially reduce the number of unique words in the vocabulary because we allow the reuse of IDs across different TPs. For example, the first method name identifier in any transformation pair will be replaced with the ID `METHOD_1`, regardless of the original method name.

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

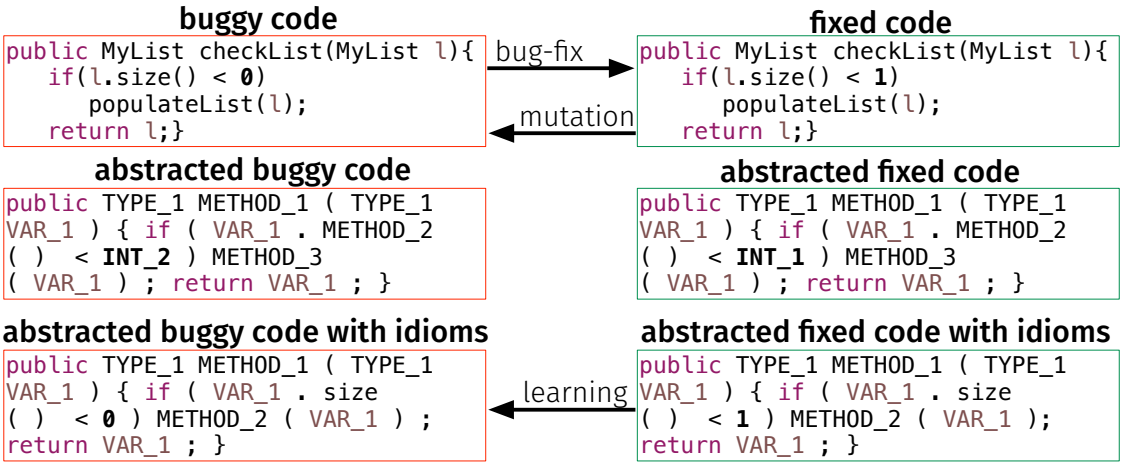

Figure 3.1: Transformation Pair Example.

Figure 3.1 shows an example of a TP. The left side is the buggy code and the right side is the same method after the bug-fix (changed the `if` condition). The abstracted stream of tokens is shown below each corresponding version of the code. Note that the fixed version is abstracted before the buggy version. The two abstracted streams share most of the IDs, except for the `INT_2` ID (corresponding to the int value 0), which appears only in the buggy version.

There are some identifiers and literals that appear so often in the source code that, for the purpose of our abstraction, they can almost be treated as keywords of the language. For example, the variables `i`, `j`, or `index` are often used in loops. Similarly, literals such as `0`, `1`, `-1` are often used in conditional statements and return values. Method names, such as `getValue`, appear multiple times in the code as they represent common concepts. These identifiers and literals are often referred to as "idioms" [55]. We keep these idioms in our representation, that is, we do not replace idioms with a generated ID, but rather keep the original text in the code representation. To define the list of idioms, we first randomly sampled 300k TPs 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 frequent words (outliers of the distribution). In particular, we focused on the top 0.005% of

the distribution. Two authors manually analyzed this list and curated a set of 272 idioms. Idioms also include standard Java types such as `String`, `Integer`, common `Exceptions`, *etc.* The complete list of idioms is available on our online appendix [43].

Figure 3.1 shows the *idiomized* abstracted code at the bottom. The method name `size` is now kept in the representation and not substituted with an ID. This is also the case for the literal values `0` and `1`, which are very frequent idioms. Note that the method name `populateList` is now assigned ID `METHOD_2` rather than `METHOD_3`. This representation provides enough context and information to effectively learn code transformations, while keeping a limited vocabulary ($|V| = \sim$430). Note that the abstracted code can be mapped back to the real source code using the the mapping (M).

3.2.2.3 Filtering Invalid TPs

Given the extracted list of 2.3M TPs, we manipulated their code via the aforementioned abstraction method. During the abstraction, we filter out such TPs that: (i) contain lexical or syntactic errors (*i.e.*, either the lexer or parser failed to process them) in either the buggy or fixed version of the code; (ii) their buggy and fixed abstracted code ($abstract_b$, $abstract_f$) resulted in equal strings. The equality of $abstract_b$ and $abstract_f$ is evaluated while ignoring whitespace, comments or annotations edits, which are not useful in learning mutants. Next, we filter out TPs that performed more than 100 atomic AST actions ($|A| > 100$) between the buggy and fixed version. The rationale behind this 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 mutations. Finally, we discard long methods and focus on small/medium size TPs. We filter out TPs whose fixed or buggy abstracted code is longer than 50 tokens. We discuss this choice in the Section 3.5 and report preliminary results also for longer methods. After the filtering, we obtained ~380k TPs.

3.2.2.4 Synthesis of Identifiers and Literals

TPs are the examples we use to make our model learn how to mutate source code. Given a $tp = \{m_b, m_f, A\}$, we first abstract its code, obtaining $tpa = \{abstract_b, abstract_f, A, M\}$. The fixed code $abstract_f$ is used as input to the model which is trained to output the corresponding buggy code (mutant) $abstract_b$. This output can be mapped back to real source code using M.

In the current usage scenario (*i.e.*, generating mutants), when the model is deployed, we do not have access to the oracle (*i.e.*, buggy code, $abstract_b$), 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 mutated 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. Synthesizing new identifiers would involve extensive knowledge about the project, control and data flow information. For this reason, we discard the TPs that contain, in the buggy method m_b, new identifiers not seen in the fixed method m_f. The rationale is that we want to train our model from examples that rearrange existing identifiers, keywords and operators to mutate source code. Instead, this is not the case for literals. While we cannot perfectly map a new literal ID to a concrete value, we can still synthesize new literals by leveraging the type information embedded in the ID. For example, the (fixed) if condition in Figure 3.1 `if(VAR_1.METHOD_2( ) < INT_1)` should be mutated in its buggy version `if(VAR_1.METHOD_2( ) < INT_2)`. The value of `INT_2` has never appeared in the input code (fixed), but we could still generate a compilable mutant by randomly generating a new integer value (different from any literal in the input code). While in these cases the literal value is randomly generated, the mutation model still provides the prediction about which literal to mutate.

For such reasons, we create two sets of TPs, hereby referred as TP_{ident} and $TP_{ident-lit}$. TP_{ident} contains all TPs $tpa = \{abstract_b, abstract_f, A, M\}$ such that every identifier ID (`VAR_`, `METHOD_`, `TYPE_`) in $abstract_b$ is found also in $abstract_f$. In this set we do allow new literal IDs (`STRING_`, `INT_`, *etc.*). $TP_{ident-lit}$ is a subset of TP_{ident}, which is more restrictive, and only contains the transformation pairs $tpa = \{abstract_b, abstract_f, A, M\}$ such that every identifier and literal ID in $abstract_b$ is found also in $abstract_f$. Therefore, we do not allow new identifiers nor literals.

The rationale behind this choice is that we want to learn from examples (TPs) where the model is able to generate compilable mutants (*i.e.*, generate actual source code, with real values for IDs). In the case of the $TP_{ident-lit}$ set, the model will learn from examples that do not introduce any new identifier and literal. This means that the model will likely generate code for which every literal and identifier can be mapped to actual values. From the set TP_{ident} the model will likely generate code for which we can map every identifier but we may need to generate new random literals.

In this context it is important to understand the role played by the idioms in our code representation. Idioms help to retain transformation pairs that we would otherwise discard, and learn transformation of literal values that we would otherwise need to randomly generate. Consider again the previous example `if(VAR_1 . METHOD_2 ( ) < INT_1)` and its mutated version `if(VAR_1 . METHOD_2 ( ) < INT_2)`. In this example, there are no idioms and, therefore, the model learns to mutate `INT_1` to `INT_2` within the `if` condition. However, when we want to map back the mutated (buggy) representation to actual source code, we will not have a value for `INT_2` (which does not appear in the input code) and, thus, we will be forced to generate a synthetic value for it. Instead, with the idiomized abstract representation the model would treat the idioms `0` and `1` as keywords of the language and learn the exact transformation of the `if` condition. The proposed mutant will therefore contain directly the idiom value (1) rather than `INT_2`. Thus, the model will learn and propose such transformation without the need to randomly generate literal values. In summary, idioms increase the number of transformations incorporating real values rather

than abstract representations. Without idioms, we would lose these transformations and our model could be less expressive.

3.2.2.5 Clustering

The goal of clustering is to create subsets of TPs such that TPs in each cluster share a similar list of AST actions. Each cluster represents a cohesive set of examples so that a trained a model can apply those actions to a new code.

As previously explained, each transformation pair $tp = \{m_b, m_f, A\}$ includes a list of AST actions A. In our dataset, we found ∼1,200 unique AST actions, and each TP can perform a different number and combination of these actions. Deciding whether the transformation pairs, tp_1 and tp_2, perform a similar sequence of actions and, thus, should be clustered together, is far from trivial. Possible similarity functions include the number of shared elements in the two sets of actions and the frequency of particular actions within the sets. Rather than defining such handcrafted rules, we choose to learn similarities directly from the data. We rely on an unsupervised learning algorithm that learns vector representations for the lists of actions A of each TP. We treat each list of AST actions (A) as a document and rely on *doc2vec* [161] to learn a fixed-size vector representation of such variable-length documents embedded in a latent space where similarities can be computed as distances. The closer two vectors are, the more similar the content of the two corresponding documents. In other words, we mapped the problem of clustering TPs to the problem of clustering continuous valued vectors. To this goal, we use k-means clustering, requiring the number of clusters (k) into which to partition the data upfront. When choosing k, we need to balance two conflicting factors: (i) maximize the number of clusters so that we can train several different mutation models and, as a consequence, apply different mutations to a given piece of code; and (ii) have enough training examples (TPs) in each cluster to make the learning possible. Regarding the first point, we target at least three mutation models. Concerning the second point, with the available TPs dataset we could reasonably train no more than six clusters, so that each of those contain enough

TPs. Thus, we experiment on the dataset $TP_{ident-lit}$ with values of k going from 3 to 6 at steps of 1 and we evaluate each clustering solution in terms of its Silhouette statistic [107, 112], a metric used to judge the quality of clustering. We found that $k = 5$ generates the clusters with the best overall Silhouette values. We cluster the dataset $TP_{ident-lit}$ into clusters: C_1, C_2, C_3, C_4, C_5.

3.2.3 Learning Mutations

3.2.3.1 Dataset Preparation

Given a set of TPs (*i.e.*, TP_{ident}, $TP_{ident-lit}$, $C_1, \ldots, C_5$) we use the instances to train our Encoder-Decoder model. Given a $tpa = \{abstract_b, abstract_f, A, M\}$ we use only the pair $(abstract_f, abstract_b)$ of fixed and buggy abstracted code for learning. No additional information about the possible mutation actions (A) is provided during the learning process to the model. The given set of TPs is randomly partitioned into: training (80%), evaluation (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).

3.2.3.2 Encoder-Decoder Model

Our models are based on an RNN Encoder-Decoder architecture, commonly adopted in Machine Translation [106, 171, 59]. This model comprises two major components: an RNN Encoder, which *encodes* a sequence of terms $\boldsymbol{x}$ into a vector representation, and an RNN Decoder, which *decodes* the representation into another sequence of terms $\boldsymbol{y}$. The model learns a conditional distribution over a (output) sequence conditioned on another (input) sequence of terms: $P(y_1, .., y_m | x_1, .., x_n)$, where n and m may differ. In our case, given an input sequence $\mathbf{x} = abstract_f = (x_1, .., x_n)$ and a target sequence $\mathbf{y} = abstract_b = (y_1, .., y_m)$, the model is trained to learn the conditional distribution: $P(abstract_b | abstract_f) = P(y_1, .., y_m | x_1, .., x_n)$, where x_i and y_j are abstracted source

tokens: Java keywords, separators, IDs, and idioms. 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 [46] which is formed by a backward and forward RNNs, which are able to create representations taking into account both past and future inputs [54]. That is, each state h_i represents the concatenation of the states produced by the two RNNs 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 practically constitutes the attention mechanism. The vector c_i is computed as a weighted average of the states in $\mathbf{h}$, as follows: $c_i = \sum_{t=1}^{n} a_{it} h_t$ 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.

3.2.3.3 Configuration and Tuning

For the RNN Cells we tested both LSTM [96] and GRU [59], founding the latter to be slightly more accurate and faster to train. Before settling on the bi-directional Encoder, we tested the unidirectional Encoder (with and without reversing the input sequence), but we consistently found the bi-directional one yielding more accurate results. Bucketing and padding was used to deal with the variable length of the sequences. We tested several combinations of the number of layers (1,2,3,4) and units (256, 512). The configuration that best balanced performance and training time was the one with 1 layer encoder, 2 layer decoder both with 256 units. We train our models for 40k epochs, which represented our empirically-based sweet spot between training time and loss function improvements. The evaluation step was performed every 1k epochs.

3.3 Experimental Design

The evaluation has been performed on the dataset of bug fixes described in Section **??** and answers three RQs.

RQ1: Can we learn how to generate mutants from bug-fixes? RQ1 investigates the extent to which bug fixes can be used to learn and generate mutants. We train models based on the two datasets: TP_{ident} and $TP_{ident-lit}$. We refer to such models with the name general models (GM_{ident}, $GM_{ident-lit}$), because they are trained using TPs of each dataset without clustering. Each dataset is partitioned into 80% training, 10% validation, 10% testing.

BLEU Score. The first performance metric we use is the Bilingual Evaluation Understudy (BLEU) score, a metric used to assess the quality of a machine translated sentence [149]. BLEU scores require reference text to generate a score, which indicates how similar the candidate and reference texts are. The candidate and reference texts are broken into n-grams and the algorithm determines how many n-grams of the candidate text appear in the reference text. We report the global BLEU score, which is the geometric mean of all n-grams up to four. To assess our mutant generation approach, we first compute the BLEU score between the abstracted fixed code ($abstract_f$) and the corresponding target buggy code. This BLEU score serves as our baseline for comparison. We compute the BLEU score between the predicted mutant ($abstract_p$) and the target ($abstract_b$). The higher the BLEU score, the more similar $abstract_p$ is to $abstract_b$, *i.e.*, the actual buggy code. To fully understand how similar our prediction is to the real buggy code, we need to compare the BLEU score with our baseline. Indeed, the input code (*i.e.*, the fixed code) provided to our model can be considered by itself as a "close translation" of the buggy code, therefore, helping in achieving a high BLEU score. To avoid this bias, we compare the BLEU score between the fixed code and the buggy code (baseline) with the BLUE score obtained when comparing the predicted buggy code ($abstract_p$) to the actual buggy code ($abstract_b$). If the BLEU score between $abstract_p$ and $abstract_b$ is higher than that

one between $abstract_f$ and $abstract_b$, it means that the model transforms the input code ($abstract_f$) into a mutant ($abstract_p$) that is closer to the buggy code ($abstract_b$) than it was before the mutation, *i.e.*, the mutation goes in the right direction. In the opposite case, the predicted code represents a translation that is further from the buggy code when compared to the original input. To assess whether the differences in BLEU scores between the baseline and the models are statistically significant, we employ a technique devised by Zhang *et al.* [203]. Given the test set, we generate $m = 2,000$ test sets by sampling with replacement from the original test set. Then, we evaluate the BLEU score on the m test sets both for our model and the baseline. Next, we compute the m deltas of the scores: $\delta_i = model_i - baseline_i$. Given the distribution of the deltas, we select the 95% confidence interval (CI) (*i.e.*, from the 2.5^{th} percentile to the 97.5^{th} percentile). If the CI is completely above or below the zero (*e.g.*, 2.5^{th} percentile > 0) then the differences are statistically significant.

Prediction Classification. Given $abstract_f$, $abstract_b$ and $abstract_p$, we classify each prediction into one of the following categories: (i) perfect prediction if $abstract_p = abstract_b$ (the model converts the fixed code back to its buggy version, thus reintroducing the original bug); (ii) bad prediction if $abstract_p = abstract_f$ (the model was not able to mutate the code and returned the same input code); and (iii) mutated prediction if $abstract_p \neq abstract_b$ AND $abstract_p \neq abstract_f$ (the model mutated the code, but differently than the target buggy code). We report raw numbers and percentages of the predictions falling in the described categories.

Syntactic Correctness. To be effective, mutants need to be syntactically correct, allowing the project to be compiled and tested against the test suite. We evaluate whether the models' predictions are lexically and syntactically correct by means of a Java lexer and parser. Perfect predictions and bad predictions are already known to be syntactically correct, since we established the correctness of the buggy and fixed code when extracting the TPs. The correctness of the predictions within the mutated prediction category is instead unknown. For this reason, we report both the overall percentage of syntactically correct

predictions as well as the mutated predictions. We do not assess the compilability of the code.

Token-based Operations. We analyzed and classified models' predictions also based on their tokens' operations, classifying the predictions into one of four categories: (i) insertion if #tokens predictions > #tokens input; (ii) changes if #tokens prediction = #tokens input AND prediction ≠ input; (iii) deletions if #tokens prediction < #tokens input; (iv) none if prediction = input. This analysis aims to understand whether the models are able to insert, change or delete tokens.

AST-based Operations. Next, we focus on the mutated predictions. These are not perfect predictions, but we are interested in understanding whether the transformations performed by the models are somewhat similar to the transformations between the fixed and buggy code. In other words, we investigate whether the model performs AST actions similar to the ones needed to transform the input (fixed) code into the corresponding buggy code. Given the input fixed code $abstract_f$, the corresponding buggy code $abstract_b$, and the predicted mutant $abstract_p$, we extract with GumTreeDiff the following lists of AST actions: $A_{f-b} = actions(abstract_f \rightarrow abstract_b)$ and $A_{f-p} = actions(abstract_f \rightarrow abstract_p)$. We then compare the two lists of actions, A_{f-b} and A_{f-p}, to assess their similarities. We report the percentage of mutated predictions whose list of actions A_{f-p} contains the same elements and frequency of those found in A_{f-b}. We also report the percentage of mutated predictions when only comparing their unique actions and disregarding their frequency. In those cases, the model performed the same list of actions but possibly in a different order, location or frequency than those which led to the perfect prediction (buggy code).

RQ2: Can we train different mutation models? RQ2 evaluates the five models trained using the five clusters of TPs. For each model, we evaluate its performance on the corresponding 10% test set using the same analyses discussed for RQ1. In addition, we evaluate whether models belonging to different clusters generate different mutants. To this aim, we first concatenate the test set of each cluster into a single test set. Then, we

feed each input instance in the test set (fixed code) to each and every mutation model $M_1, .., M_5$, obtaining five mutant outputs. After that, we compute the number of unique mutants generated by the models. For each input, the number of unique mutants ranges from one to five depending on how many models generate the same mutation. We report the distribution of unique mutants generated by the models.

RQ3: What are the characteristics of the mutants generated by the models? RQ3 qualitatively assesses the generated mutants through manual analysis. We first discuss some of the perfect predictions found by the models. Then, we focus our attention on the mutated predictions (neither perfect nor bad predictions). We randomly selected a statistically significant sample from the mutated predictions of each cluster-model and manually analyzed them. The manual evaluation assesses (i) whether the functional behavior of the generated mutant differs from the original input code; (ii) the types of mutation operations performed by the model in generating the mutant.

Three judges, among the authors, were involved in this analysis, and we required each instance to be independently evaluated by two judges. The judges were presented with the original input code and the mutated code. The judges defined the mutation operations types in an open-coding fashion. Also, they were required to indicate whether the performed mutation changed the code behavior. After the initial evaluation, two of the three judges met to discuss and resolve the conflicts in the evaluation results. We define a conflict as any instance for which two of the judges disagreed on either the changes in the functional behavior or the set of mutation operations assigned. We report the distribution of the mutation operators applied by the different cluster-models and highlight the differences.

3.4 Results

RQ1: Can we learn how to generate mutants?

Table 3.1: BLEU Score

Model	$abstract_f$ - $abstract_b$ (baseline)	$abstract_p$ - $abstract_b$ (mutation)	2.5^{th} percentile δ =mutation - baseline
GM_{ident}	71.85	76.68	+5.63
$GM_{ident-lit}$	70.07	76.92	+7.97
M_1	67.18	82.16	+17.01
M_2	51.58	50.96	+1.01
M_3	81.89	83.15	+0.94
M_4	67.04	78.87	+12.45
M_5	65.68	77.73	+13.51

BLEU Score. The top part of Table 3.1 shows the BLEU scores obtained by the two general models and compared with the baseline. The rightmost column represents the 2.5^{th} percentile of the distribution of the deltas. Compared to the baseline, the models achieve a better BLEU score when mutating the source code w.r.t. the target buggy code. The differences are statistically significant, and the 2.5^{th} percentile of the distribution of the deltas (+5.63 and +7.97), shows that the models' BLEU scores are significantly higher than those obtained by the baseline. The observed increase in BLEU score indicates that the code mutated by our approach ($abstract_p$) is more similar to the buggy code ($abstract_b$) than the input code ($abstract_f$). Thus, the injected mutations push the fixed code towards a "buggy" state, exactly what we expect from mutation operators. While our baseline is relatively simple, improvements of few BLEU score points have been treated as "considerable" in neural machine translation tasks [199].

Prediction Classification. Table 3.2 shows the raw numbers and percentages of predictions falling into the three categories previously described (*i.e.*, perfect, mutated, and bad predictions). The GM_{ident} generated 1,991 (17%) perfect predictions whereas $GM_{ident-lit}$ 2,132 (21%) perfect predictions. We fed into the trained model a fixed piece of code, which the model has never seen before, and the model was able to perfectly predict the buggy version of that code, *i.e.*, to replicate the original bug. No information about the type of mutation operations to perform nor mutation locations are provided to the model. The fixed code is its only input. It is also important to note that, for the perfect predictions of the $GM_{ident-lit}$ model, we can transform the entire abstracted code to actual source

Table 3.2: Prediction Classification

Model	Perfect pred.	Mutated pred.	Bad pred.	Total
GM_{ident}	1,991 (17%)	6,020 (52%)	3,548 (31%)	11,559
$GM_{ident-lit}$	2,132 (21%)	5,240 (52%)	2,644 (27%)	10,016
M_1	1,348 (45%)	1,500 (49%)	190 (6%)	3,038
M_2	65 (9%)	635 (91%)	1 (0%)	701
M_3	392 (13%)	967 (33%)	1,603 (54%)	2,962
M_4	721 (29%)	1,453 (57%)	358 (14%)	2,532
M_5	366 (34%)	681 (63%)	33 (3%)	1,080

code by mapping each and every ID to their corresponding value. The perfect predictions generated by GM_{ident} can be mapped to actual source code but, in some cases, we might need to randomly generate new literal values.

GM_{ident} and $GM_{ident-lit}$ generate 6,020 (52%) and 5,240 (52%) mutated predictions, respectively. While these predictions do not match the actual buggy code, they still represent meaningful mutants. We analyze these predictions in terms of syntactic correctness and types of operations they perform.

Finally, GM_{ident} and $GM_{ident-lit}$ are not able to mutate the source code in 3,548 (30%) and 2,644 (26%) cases, respectively. While the percentages are non-negligible, it is still worth noting that overall, in 69% and 73% of cases, the models are able to mutate the code. These instances of bad predictions can be seen as cases where the model is unsure on how to properly mutate the code. There are different strategies that could be adopted to force the model to mutate the code (*e.g.*, penalize during training predictions that are equal to the input code, modify the inference step, or using beam search and select the prediction that is not equal to the input).

Syntactic Correctness. Table 3.3 reports the percentage of syntactically correct predictions performed by the model. More than 98% of the model predictions are lexically and syntactically correct. When focusing on mutated predictions, the syntactic correctness is still very high (>96%). This indicates that the model is able to learn the correct syntax rules from the abstracted representation we use as input/output of the model. While we

Table 3.3: Syntactic Correctness

Model	Mutated pred.	Overall
GM_{ident}	96.96%	98.42%
$GM_{ident-lit}$	96.56%	98.20%
M_1	96.07%	98.06%
M_2	95.12%	95.58%
M_3	94.42%	98.18%
M_4	95.25%	97.27%
M_5	91.48%	94.63%

do not report statistics on the compilability of the mutants, we can assume that the ∼20% perfect predictions generated by the models are compilable, since they correspond to actual buggy code that was committed to software repositories. This means that the compilability rate of the mutants generated by our models is at least around 20%. This is a very conservative estimation that does not consider the mutated predictions. Brown *et al.* [55] achieved a compilability rate of 14%. Moreover, "the majority of failed compilations (64%) arise from simple parsing errors" [55], whereas we achieve a better estimated compilability and a high percentage of syntactically correct predictions.

Token-based Operations. Table 3.4 shows the classification of predictions based on the token-based operations performed by the models. GM_{ident} and $GM_{ident-lit}$ generated predictions that resulted in the insertion of tokens in 1% of the cases, changed nodes in 5% and 3% of the cases, and deletion of tokens in 63% and 69%, respectively. While most of the predictions resulted in token deletions, it is important to highlight that our models are able to generate predictions that insert and change tokens. We investigated whether these results were in-line with the actual data, or whether this was due to a drawback of our learning. We found that the operations performed in the bug-fixes we collected are: 72% insertion, 8% deletion, and 20% changes. This means that bug-fixes mostly tend to perform insert operations (*e.g.*, adding an `if` statement to check for an exceptional condition), which means that when learning to inject bugs by mutating the code, it is expected to observe a vast majority of delete operations (see Table 3.4).

Table 3.4: Token-based Operations

Model	Insertion	Changes	Deletion	None
GM_{ident}	97 (1%)	624 (5%)	7,290 (63%)	3,548 (31%)
$GM_{ident-lit}$	125 (1%)	264 (3%)	6,983 (70%)	2,644 (26%)
M_1	11 (0%)	30 (1%)	2,807 (93%)	190 (6%)
M_2	27 (4%)	11 (2%)	662 (94%)	1 (0%)
M_3	42 (2%)	217 (7%)	1,100 (37%)	1,603 (54%)
M_4	87 (3%)	123 (5%)	1,964 (78%)	358 (14%)
M_5	25 (2%)	20 (2%)	1,002 (93%)	33 (3%)

Table 3.5: AST-based Operations

Model	Same Operation Set	Same Operation List
GM_{ident}	16.02%	13.90%
$GM_{ident-lit}$	24.44%	21.90%
M_1	54.46%	48.66%
M_2	11.18%	10.23%
M_3	15.20%	14.27%
M_4	31.65%	29.24%
M_5	41.55%	37.44%

AST-based Operations. Table 3.5 reports the percentage of mutated predictions that share the same set or list of operations that would have led to the actual buggy code. GM_{ident} and $GM_{ident-lit}$ generate a significant amount of mutated predictions which perform the same set (16% and 24% respectively) or the same type and frequency (14% and 21%) of operations w.r.t. the buggy code. This shows that our models can still generate mutated code that is similar to the actual buggy code.

Summary for RQ$_1$. Our models are able to learn from bug-fixes how to mutate source code. The general models generate mutants that perfectly correspond to the original buggy code in ∼20% of the cases. The mutants generated are mostly syntactically correct (>98%) and with an estimated compilability rate of at least 20%.

RQ2: Can we train different mutation models? We present the performance of the cluster models M_1,..,M_5 based on the metrics introduced before. Each model has

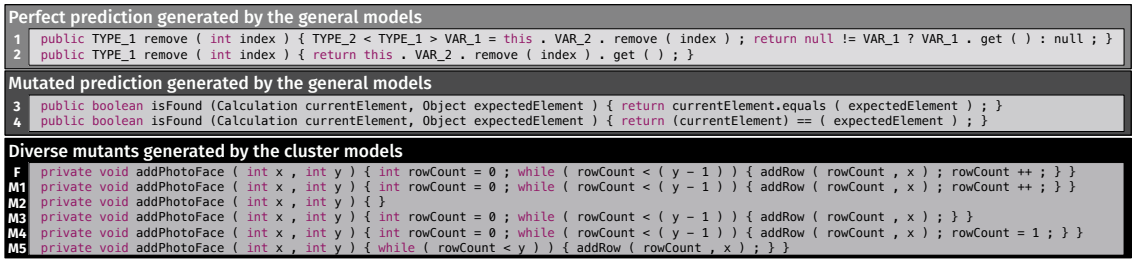

Figure 3.2: Qualitative Examples

been trained and evaluated on the corresponding cluster of TPs, with respective sizes of $C_1 = 30{,}385$, $C_2 = 7{,}016$, $C_3 = 29{,}625$, $C_4 = 25{,}320$, and $C_5 = 10{,}798$.

BLEU Score. Table 3.1 shows the BLEU scores obtained by the five models. The BLEU scores for these models (mutation column) are relatively high, between 77.73 and 83.15 (with exception of model M_2), meaning that the mutated code generated by such models is a very close translation of the actual buggy code. Looking at the distribution of the deltas, we can notice that all the 2.5^{th} percentiles are greater than zero, meaning that the models achieve a BLEU score which is statistically better than the baselines. Even in the case of M_2, for which the global BLEU score is slightly lower than the baseline, when the comparison is performed over 2,000 random samples, it outperforms the baseline.

Prediction Classification. Table 3.2 shows the raw numbers and percentages of predictions falling in the three categories we defined. Model M_1 achieves the highest percentage of Perfect predictions (44%), followed by model M_5 (33%) and model M_4 (28%). This means that, given a fixed code, it is very likely that at least one of the models would predict the actual buggy code, as well as other interesting mutants. At the same time, the percentages of Bad predictions decreased significantly (except for M_3) w.r.t. the general models.

The high percentage of bad predictions for M_3 can be partially explained by looking at the actual data in the cluster. The TPs in C_3 exhibits small transformations of the code. This is also noticeable from Table 3.1, which shows a baseline BLEU score of 81.89 (the highest baseline value), which means that the input fixed code is already a close translation of the corresponding buggy code. This may have led the model to fall in a *local*

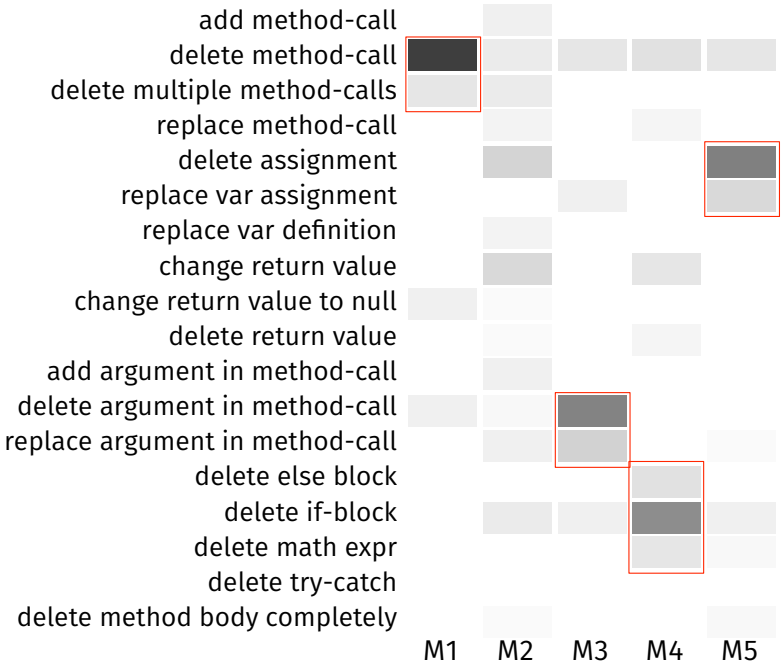

Figure 3.3: Cluster Models Operations

minimum where the mutation of the fixed code is the fixed code itself. Solutions for this problem may include: (i) further partitioning the cluster into more cohesive sub-clusters; (ii) allowing more training times/epochs for such models; (iii) implementing changes in the training/inference that we discussed previously.

Syntactic Correctness. Table 3.3 reports the percentage of syntactically correct predictions performed by the models. Overall, the cluster model results are in-line with what was found for the general models, with an overall syntactic correctness between 94.63% and 98.18%. When focusing only on the mutated predictions, we still obtain very high syntactic correctness, between 91% and 97%. In terms of compilability, we could expect even better results for these models, given the higher rate of perfect predictions (which are likely to be compilable) generated by the cluster models.

Token-based Operations. Table 3.4 shows the classification of predictions based on the token-based operations performed by the models. The results for the cluster models are similar to what we found for general models, with higher percentages of deletions. In the

next sections we will look more into the differences among the operations performed by each model.

AST-based Operations. Table 3.5 reports the percentage of mutated predictions sharing the same set or list of operations w.r.t. the target buggy code. Cluster models, trained on cohesive sets of examples, generate a higher percentage of mutated predictions sharing the same set or list of operations as the target buggy code, as compared to the general models. The results for M_1, M_4, and M_5 are particularly good as they generate mutants with the same set of operations in 54%, 31%, and 41% of the cases, respectively, and with the same list of operations in 48%, 29%, and 37%, respectively.

Unique Mutants Generated. The distribution of unique mutants generated by the five models has the 1st Qu. and Median values equal to 4, the mean equal to 4.2, and the 3rd Qu. equal to 5. Thus, the distribution appear to be skewed towards the maximum value (5). This demonstrates that we are able to train different mutation models that generate diverse mutants given the same input code.

Generate Multiple Mutants. We showed that clusters models are able to generate a diverse set of mutants, however it is also possible – for each single model – to generate k different mutants for a given piece of code via beam search decoding. In a preliminary investigation we performed, we found that each model can generate more than 50 diverse mutants for a single method, with impressive ∼80% syntactic correctness.

Summary for RQ$_2$. The cluster models generate a high percentage of perfect predictions (between 9% and 45%) with a syntactic correctness between 94% and 98%. Even when the models generate mutants that are not perfect predictions, they usually apply a similar set of operations w.r.t. the buggy code. Furthermore, the trained models generate diverse mutants.

RQ3: What are the characteristics of the mutants generated by the models? Figure 3.2 shows examples of perfect and mutated predictions generated by the general models, as well as diverse mutants generated by the cluster models for the same input

code. At the top, each example shows the input code (fixed) followed by the generated mutated code.

The first example is a perfect prediction generated by the general model. The top line is the abstracted fixed code fed to the model, and the bottom line represents the output of the model, which perfectly corresponds to the target buggy code. The fixed code first removes the element at `index` from `VAR_2`, assigning it to the `VAR_1`, and then, if the newly defined variable is not null, it invokes the method `get` and returns its value, otherwise it returns null. The general model was able to apply different transformations of the code to generate the original buggy code, which invokes all the methods in sequence and returns the value. If the removed element is null, the buggy code will throw an exception when invoking the method `get`. This transformation of the code does not fit in any existing mutation operator category.

Next, we report an interesting case of mutated prediction. In this case, we used the mapping M to automatically map back every identifiers and literals, showing the ability to generate *actual source code* from the output of the model. The model replaced the `equals()` method call with an equality expression (`==`). This shows how the model was able to learn common bugs introduced by developers when comparing objects. Note that the method name `equals` is an idiom, which allowed the model to learn this transformation.

Finally, the bottom part of Figure 3.2 shows the five mutants generated by the cluster models for the same fixed code (F) provided as input. In this case, we used the mapping M to retrieve the source code from the output of the models. We selected this example because it shows both interesting mutations and some limitations of our approach. M_1 was not able to generate a mutant and returned the same input code (bad prediction). M_2 generated a mutant by removing the entire method body. While this appears like a trivial mutation, it is still meaningful as the method is not supposed to return a value, but only perform computations that will result in some side-effects in the class. This means that the test suite should carefully evaluate the state of the class after the invocation of the mutant. Mutants generated by M_3 and M_4 are the most interesting. They both introduce

an infinite-loop, but in two different ways. M_3 deletes the increment of the `rowCount` variable, whereas M_4 resets its value to 1 at each iteration. Finally, M_5 changes the `if` condition and introduces an infinite loop similarly to the model M_3. However it also deletes the variable definition statement for `rowCount`, making the mutant not compilable. All the predictions (including perfect, mutated, and diverse) are available in our appendix [43].

In the manual evaluation, three judges analyzed a total of 430 samples (90, 82, 86, 89, and 83 from M_1, M_2, M_3, M_4, M_5, respectively). For every sample instance, the judges agreed that the mutated code appeared to have a different functional behavior w.r.t. the original input code. Only one case was debated, corresponding to a mutant which was created by deleting a print call. In this case, the *functional* behavior may or may not have been changed, depending on whether the output console is assessed as part of the behavior of the method. Thus, all the instances except one were evaluated as actual mutants that introduced a buggy behavior.

Figure 3.3 shows a heatmap of the frequency of mutation operations for each trained model. The intensity of the color represents the frequency with which a particular operation (specified by the row) was performed by the particular cluster model (columns). Due to space constraints, the rows of the heatmap contain only a subset of the 85 unique types of operations performed by the models, *i.e.*, only those performed in at least 5% of the mutations by at least one model.

We also highlighted in red boxes the peculiar, most frequent operations performed by each model. M_1 appears to focus on deletion of method calls; M_3 on deletion and replacement of an argument in a method call; M_4 mostly operates on `if-else` blocks and its logical conditions; M_5 focuses on deleting and replacing variable assignments. Finally, it is worth noting the large variety of operations performed by M_2, ranging from addition, deletion, and replacement of method calls, variable assignments, arguments, *etc.*. This might also explain the lower BLEU score achieved by the latter model, which performs large and more complex operations w.r.t. the other models which tend to focus on a smaller set of operations. Differences among the mutation models can also be appreciated by the

number of different mutation operations performed for each mutant. The models M_1, M_2, M_3, M_4, M_5 performed 1.19, 3.48, 1.42, 1.93, 2.02 average number of operations for each mutant, respectively.

Summary for RQ$_3$. The mutation models are capable of performing a diverse set of operations to mutate source code.

3.5 Threats to Validity

Construct validity. To have enough training data, we mined bug-fixes in GitHub, rather than using curated bug-fix datasets such as Defects4j, while still very useful but limited in size. To mitigate imprecisions in our datasets (*i.e.*, commits not related to bug fixes), we manually analyzed a sample of the extracted commits. Moreover, we disregarded large commits (too many files/AST operations) that might refer to tangled changes.

Internal validity. In assessing whether the generated mutants change the behavior of the code, we analyzed the mutated method in isolation (*i.e.*, not in the context of its system). This might have introduced imprecisions that were mitigated by assigning multiple evaluators to the analysis of each mutant.

External validity. We only focused on Java code. However, the learning process is language-independent and the whole infrastructure can be instantiated for different languages by replacing the lexer, parser and AST differencing tools. We only focused on methods having no more than 50 tokens. In our appendix [43] we report experimental results on larger methods (between 50 and 100 tokens) using the same configuration of the network and training epochs. The trained model was still able to generate ∼6% of perfect predictions. We are confident that with more training time and parameters' tuning better results can be obtained for larger methods.

3.6 Related Work

Brown *et al.* [55] leveraged bug-fixes to extract syntactic-mutation patterns from the diffs of patches. Our approach is novel and differs from Brown *et al.* in several aspects:

- Rather than extracting all possible mutation operators from syntactic diffs, we automatically learn mutations from the data;
- Rather than focusing, in isolation, on contiguous lines of code changed in the diff, we learn the mutation in its context (*e.g.*, method). This allows learning which type and variation of mutation operator is more likely to be effective, given the current context (*i.e.*, methods, variables, scopes and blocks);
- Our approach is able to automatically mutate identifiers and literal by inserting idioms (based on what learned) in the new mutant. When the model suggests to mutate a literal with another unknown literal, it is generated randomly. Brown's *et al.* approach does not contemplate the synthesis of new identifiers (cfr. Section 2.3 [55]);
- Rather than extracting a single mutation pattern, we can learn co-occurrences and combinations of multiple mutations;
- While the approach by Brown *et al.* randomly applies mutation operators to any code location unless the user specifies a rule for that, our approach automatically applies, for a given piece of code, the mutation(s) that according to the learning might reflect likely bugs occurring in such a location. While limiting mutants only to the most suitable ones for each location might not be always necessary, because one can apply as many mutants as possible to increase fault detection, this could lead to an overestimate of at test suite effectiveness or to more effort to unnecessarily augment a test suite. In a view of test suite optimization, an approach that learns where and how to mutate code is therefore desirable.

Currently, a direct comparison between the two approaches was not viable as the approaches have been developed for different programming languages (C *vs.* Java).

Different general-purpose mutation frameworks have been defined in the literature, including μJava [129], Jester [31], Major [104], Jumble [184], PIT [32], and javaLanche [164]. The main novelty of our work over those approaches is the automation of the learning and application of the mutation. Relevant to our work are also studies investigating the relationship between mutants and real faults. Andrews *et al.* [41, 42] showed that carefully selected mutants can provide a good assessment of a test suite's ability to catch real faults and hand-seeded faults can underestimate the test suite's bug detection capability. Daran and Thévenod-Fosse [63] found that the set of errors produced by carefully selected mutants and real faults with a given test suite are quite similar, while Just *et al.* [105] reported that some types of real faults are not coupled to mutants and highlighted the need for new mutation operators. Finally, Chekham *et al.* [57] showed that strong mutation testing yields high fault revelation, while this is not the case for weak mutation testing. Our work builds on these studies, cementing an approach for learning mutants from real bug fixes. Hence, we avoid the need for manually selecting the mutants to inject and increase the chances of generating mutants representative of real bugs.

Allamanis *et al.* [37] generate tailored mutants, *e.g.*, exploiting API calls occurring elsewhere in the project and show that tailored mutants are well-coupled to real bugs. Differently from them, we automatically learn how to mutate code from an existing dataset of bugs rather than using heuristics.

3.7 Conclusion

We presented the first approach to automatically learn mutants from existing bug fixes. The evaluation we performed highlights that the generated mutants are similar to real bugs, with 9% to 45% of them (depending on the model) reintroducing in the fixed code (provided as input) the actual bug. Moreover, our models are able to learn the *correct*

code syntax, without the need for syntax rules as input. We release the data and code to allow researchers to use it for learning other transformations of code [43]. Future work includes additional fine tuning of the RNN parameters to improve performance.

Chapter 4

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

4.1 Introduction

Localizing and fixing bugs is known to be an effort-prone and time-consuming task for software developers [103, 165, 194]. To support programmers in this common activity, researchers have proposed a number of approaches aimed at automatically repairing programs [45, 119, 78, 79, 117, 109, 126, 191, 144, 134, 108, 201, 133, 172, 116, 52, 156, 153, 193, 170]. The proposed techniques either use a generate-and-validate approach, which consists of generating many repairs (*e.g.*, through Genetic Programming like GenProg [192, 121]), or an approach that produces a single fix [144, 102]. While automated program repair techniques still face many challenges to be applied in practice, existing work has made strides to be effective in specific cases. These approaches, given the right circumstances, substantially contribute in reducing the cost of bug-fixes for developers [120, 130].

Two major problems automated repair approaches have, is producing patches acceptable for programmers and especially for generate-and-validate techniques, over-fitting

patches to test cases. Qi *et al.* [157] 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.* [117] 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 [126], 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.* [117] 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 [158], defect prediction [189], bug localization [115], clone detection [196], code search [81], learning API sequences [83], or recommending method names [35].

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 [181] 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[202, 168]. 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 [67]. 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 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 chapter provides the following contributions:

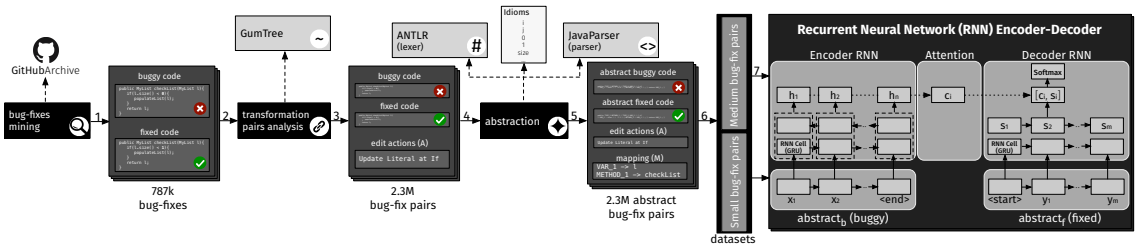

Figure 4.1: Overview of the process used to experiment with an NMT-based approach.

- 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 [43].

4.2 Approach

Fig. 4.1 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 [80] (Section 4.2.1). From the bug-fixes, we extract method-level pairs of *buggy* and corresponding *fixed* code named *bug-fix pairs* (BFPs) (Section 4.2.2.1). BFPs are the examples that we use to learn how to fix code from bug-fixes ($buggy \rightarrow fixed$). We use GumTree [67] 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 4.2.2.2) 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 4.2.2.3). Finally, for each set, we use an encoder-decoder model to learn how to transform a *buggy* code into a corresponding *fixed* version (Section 4.2.3). The trained models can be used to generate patches for unseen buggy code.

4.2.1 Bug-Fixes Mining

We downloaded from GitHub Archive [80] 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 [69]: ("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 [44, 93], 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 [43].

For each bug-fixing commit, we extracted the source code before and after the bug-fix using the GitHub Compare API [75]. 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.

4.2.2 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.

4.2.2.1 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 [67] to compute the AST differencing between f_b and f_f. This computes the sequence of edit actions performed at the AST level that allows to transform the f_b's AST into the f_f's AST.

Since the file-level granularity could be too large to learn patterns of transformation, we separate the code into method-level fragments that will constitute our BFPs. The rationale for choosing method-level BFPs is supported by several reasons. First, methods represent a reasonable target for fixing activities, since they are likely to implement a single task or functionality. Second, methods provide enough meaningful context for learning fixes, such as variables, parameters, and method calls used in the method. This choice is justified by recent empirical studies, which indicated how the large majority of fixing patches consist of single line, single churn or, worst cases, churns separated by a single line [169]. 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 [113, 33].

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

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

4.2.2.2 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 [152, 150], tokenizes the raw code into a stream of tokens, that is then fed into a Java parser [185], 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" [55]. 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 [43].

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

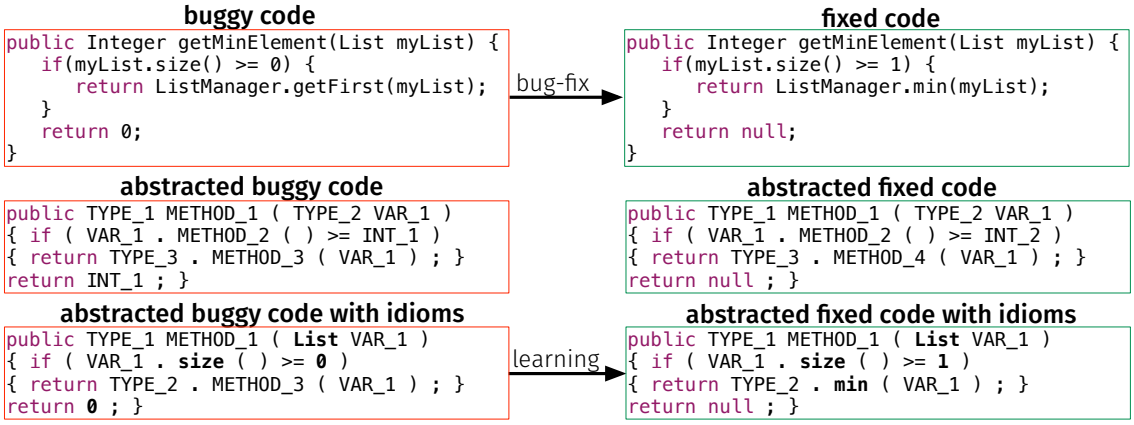

Figure 4.2: Code Abstraction Example.

To better understand our representation, let us consider the example in Fig. 4.2, where we see a bug-fix related to finding the minimum value in a list of integers. The buggy method contains three errors, which the fixed code rectifies. The first bug is within the if-condition, where the buggy method checks if the list size is greater than or equal to `0`. This is problematic since a list without any values cannot have a minimum value to return. The second bug is in the method call `getFirst`, this will return the first element in the list, which may or may not be the minimum value. Lastly, if the if-condition fails in the buggy method then the method returns `0`; returning `0` when the minimum is unable to be identified is incorrect as it indicates that one of the elements within the list is `0`. The fixed code changes the if-condition to compare against a list size of `1` rather than `0`, uses the `min` method to return the minimum value and changes the return value to `null` when the if-condition fails.

Using the buggy and fixed code for training, although a viable and realistic bug-fix, presents some issues. When we feed the buggy piece of code to the Java Parser and Lexer, we identify some problems with the mapping. For example, the abstracted fixed

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

4.2.2.3 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. 4.3. 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.

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. 4.3 it is clear that most of the data points are concentrated in the interval 0-100. Further

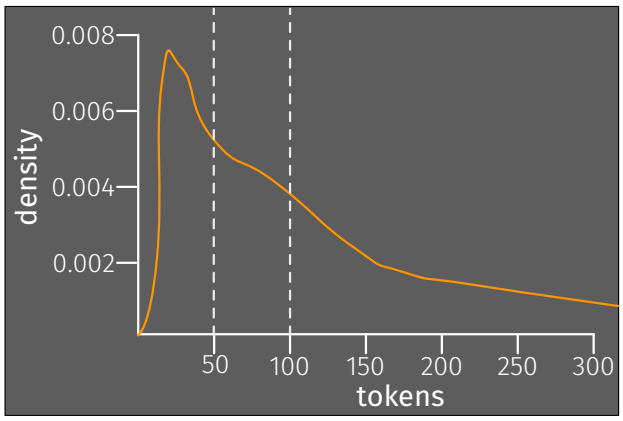

Figure 4.3: Distribution of BFPs by the number of tokens.

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 $BFP_{medium} = \{50 < bfp \leq 100\}$.

4.2.2.4 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. 4.2).

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

4.2.3 Learning Patches

4.2.3.1 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.

4.2.3.2 NMT

The experimented models are based on an RNN Encoder-Decoder architecture, commonly adopted in NMT [106, 171, 59]. This model consists of two major components: an RNN Encoder, which *encodes* a sequence of terms $\boldsymbol{x}$ into a vector representation, and an RNN Decoder, which *decodes* the representation into another sequence of terms $\boldsymbol{y}$.

The model learns a conditional distribution over a (output) sequence conditioned on another (input) sequence of terms: $P(y_1, .., y_m|x_1, .., x_n)$, where n and m may differ. In our case, given an input sequence $\mathbf{x} = abstract_b = (x_1, .., x_n)$ and a target sequence $\mathbf{y} = abstract_f = (y_1, .., y_m)$, the model is trained to learn the conditional distribution: $P(abstract_f|abstract_b) = P(y_1, .., y_m|x_1, .., x_n)$, where x_i and y_j are abstracted source tokens: Java keywords, separators, IDs, and idioms. Fig. 4.1 shows the architecture of the Encoder-Decoder model with attention mechanism [46, 128, 54]. 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 [46], which is formed by a backward and a forward RNN, which are able to create representations taking into account both past and future inputs [54]. That is, each state h_i represents the concatenation (dashed box in Fig. 4.1) 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.

4.2.3.3 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.

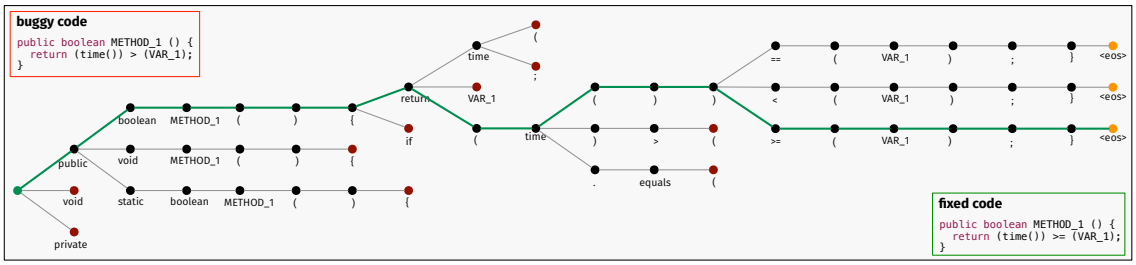

Figure 4.4: Beam Search Visualization.

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 [158].

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.

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

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. 4.4 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).

4.2.3.4 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 [96] and GRU [59]), 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 [43].

4.2.3.5 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.

4.3 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 4.2).

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

4.3.1 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 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.

4.3.2 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, 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

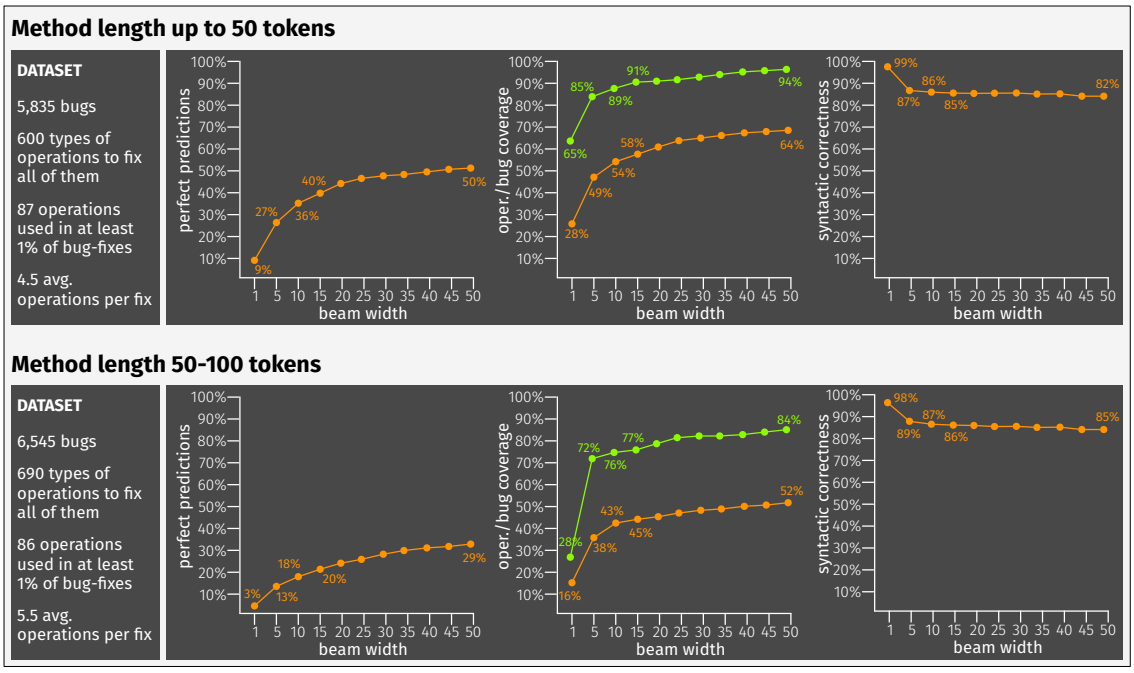

Figure 4.5: Number of perfect prediction, operation (orange) bug (green) coverage, and syntactic correctness for varying beam width and for different method lengths.

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.

4.3.3 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.

4.4 Results

4.4.1 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 [96] RNN cells. We trained the M_{small} and M_{medium} models for 50k and 55k epochs, respectively.

Table 4.1 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 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. 4.5 shows 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 [43]. The differences in performances between the M_{small} and M_{medium} could be

Table 4.1: Models' Performances

Beam	M_{small}	M_{medium}
1	538 / 5835 (**9.22**%)	211 / 6545 (**3.22**%)
5	1595 / 5835 (**27.33**%)	859 / 6545 (**13.12**%)
10	2119 / 5835 (**36.31**%)	1166 / 6545 (**17.82**%)
15	2356 / 5835 (**40.37**%)	1326 / 6545 **(20.25**%)
20	2538 / 5835 (**43.49**%)	1451 / 6545 (**22.16**%)
25	2634 / 5835 (**45.14**%)	1558 / 6545 (**23.80**%)
30	2711 / 5835 (**46.46**%)	1660 / 6545 (**25.36**%)
35	2766 / 5835 (**47.40**%)	1720 / 6545 (**26.27**%)
40	2834 / 5835 (**48.56**%)	1777 / 6545 (**27.15**%)
45	2899 / 5835 (**49.68**%)	1830 / 6545 (**27.96**%)
50	2927 / 5835 (**50.16**%)	1869 / 6545 (**28.55**%)

explained by the fact that larger methods have potentially more faulty locations where a transformation of the code could be performed.

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 width). These results indicate that Neural Machine Translation is a viable approach for learning how to fix code.

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

Fig. 4.5 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, *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. 4.5).

4.4.2.1 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.

4.4.2.2 AST Operations

The center graphs in Fig. 4.5 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.

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.

4.4.2.3 Qualitative Examples

Fig. 4.6 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. 4.6 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

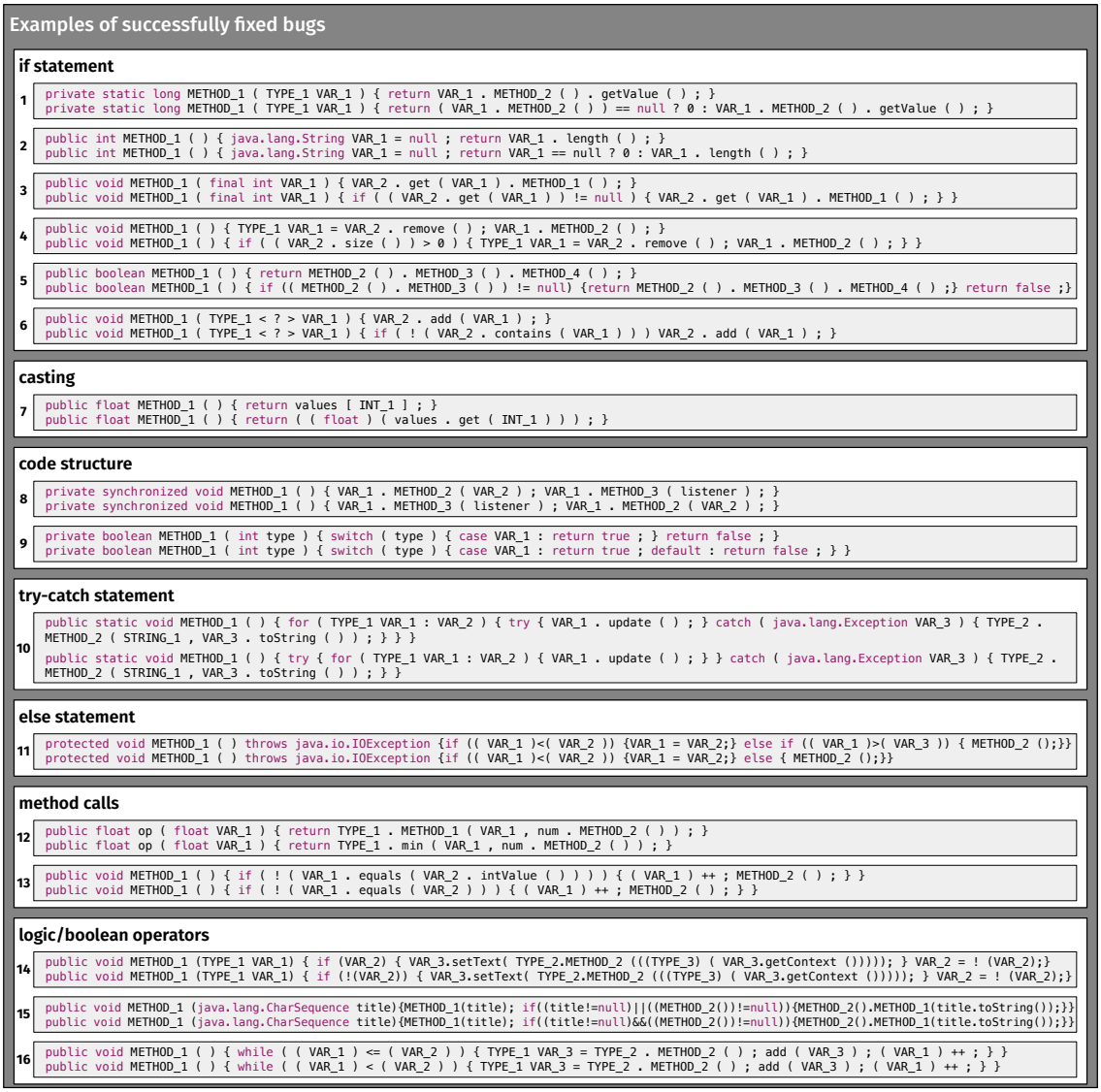

Figure 4.6: Examples of successfully-generated patches.

`if` checks to ensure the variable is not null, if it is then the method returns 0, otherwise, the method returns the result of `VAR_1.length()`. 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. 4.6). 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. 4.6). 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. 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, texttttVAR_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.* [157] 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 [43] contains many more examples of bug-fixes using different operations, considering methods with different lengths, and using a variety of beam widths.

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.

4.4.3 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. 4.7 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

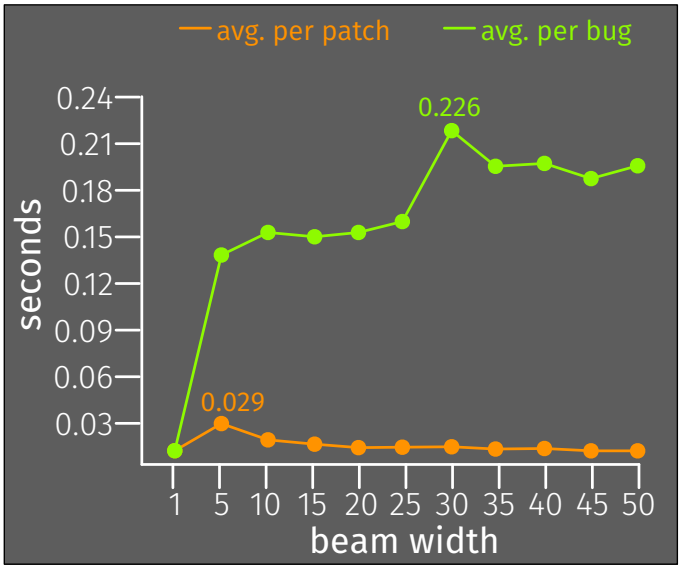

Figure 4.7: Inference Time (M_{medium}).

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 [43].

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.

4.5 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 [104] or IntroClass[118], 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 4.2 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.

4.6 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.

4.6.1 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 [136]. Behavioral repair techniques in particular change the behavior of a program under repair by changing

its source or binary code [136]. These techniques [119, 78, 167] 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 [132, 48]. Martinez *et al.* [132] 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.* [48] 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 [48]. Their findings align with Martinez *et al.* [132] 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 [154, 73, 95, 56, 143, 132, 48]. For example, GenProg [119, 78, 79] searches for statement-level modifications to make to an abstract syntax tree. The approach by Arcuri and Yao [45] co-evolves programs and test cases using a model similar to the predatory-prey one. Weimer *et al.* [191] perform program repair using a deterministic search, reducing the search space with program equivalence analysis. Le *et al.* [117] 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 [144, 134, 108, 201, 133, 172, 116, 153].

Some approaches perform automated program repair by searching the fix among manually-written patterns, as in the case of PAR [109], or through a SMT (satisfiability modulo theories)-based semantic code search over a large body of existing code, as for SearchRepair [108]. Instead, Prophet [126] 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 [52, 156]. This goal has been achieved successfully in contexts such as in massively open online courses (MOOC), where the programs are generally small and synthetic [87].

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 [181]. While our previous paper mainly presented the idea and assessed its overall feasibility, this chapter 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.

4.6.2 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 [111] 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 [204]. Nguyen *et al.* [140, 141, 142] 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 [85] 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.

4.7 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.

Chapter 5

On Learning Meaningful Code Changes via Neural Machine Translation

5.1 Introduction

Several works recently focused on the use of advanced machine learning techniques on source code with the goal of (semi)automating several non-trivial tasks, including code completion [197], generation of commit messages [101], method names [36], code comments [198], defect prediction [190], bug localization [114] and fixing [181], clone detection [196], code search [82], and learning API templates [84].

The rise of this research thread in the software engineering (SE) community is due to a combination of factors. First, is the vast availability of data and, specifically, of source code and its surrounding artifacts in open-source repositories. For instance, at the time of writing this chapter, GitHub alone hosted 100M repositories, with over 200M merged pull requests (PRs) and 2B commits. Second, DL has become a useful tool due to its ability to learn categorization of data through the hidden layer architecture making it especially proficient in feature detection [50]. Specifically, Neural Machine Translation

(NMT) has become a premier method for the translation of different languages, surpassing that of human interpretation [199]. A similar principle applies to "translating" one piece of source code into another. Here, the ambiguity of translating makes this method extremely versatile: One can learn to translate buggy code into fixed code, English into Spanish, Java into C, *etc.* Third, the availability of (relatively) cheap hardware able to efficiently run DL infrastructures.

Despite all the work done, only a few approaches have been proposed to automate non-trivial coding activities. In particular, Tufano *et al.* [181] showed that DL can be used to automate bug-fixing activities. However, there is still a lack of empirical evidence about the types of code changes that can actually be learned and automatically applied by using DL. Also, while most of the works applying DL in the software engineering field focus on quantitatively evaluating the performance of the devised technique (*e.g.*, How many bugs is our approach able to fix?), little qualitative analysis has been done to deeply investigate the meaningfulness of the output produced by DL-based approaches.

In this chapter, we make a first empirical step in the direction of quantitatively and qualitatively investigating the ability of a NMT model to learn how to automatically apply code changes just as developers do this in PRs. In particular, we harness the power of NMT to automatically "translate" a code component from its state *before* the implementation of the PR and *after* the PR has been reviewed and merged, thereby, emulating the combination of code changes that would be implemented by developers in real PRs.

We mine three large Gerrit [18] code review repositories, namely Android [15], Google Source [16], and Ovirt [17]. In total, these repositories host code reviews related to 339 sub-projects. We collected from these projects 30,292 merged PRs that underwent code review. We only considered merged and reviewed PRs for three reasons. First, we wanted to ensure that an NMT model is learning *meaningful changes*, thus, justifying the choice of mining "reviewed PRs" as opposed to any change committed in the versioning system. Second, given the deep qualitative focus of our study (details follow), we wanted to analyze the discussions carried out in the code review process to better understand the types of

changes learned by our approach. Indeed, while in the case of commits we would only have commit notes accompanying them, with a reviewed PR we can count on a rich amount of qualitative data explaining the rationale behind the implemented changes. Third, we only focus on merged PRs, since the code before and after (*i.e.*, merged) the PR is available. This is not the case for abandoned PRs. We extract method-level AST edit operations from these PRs using fine-grained source code differencing [68]. This resulted in 239,522 method pairs, each of them representing the method before (PR not submitted) and after (PR merged) the PR process. An Encoder-Decoder Recurrent Neural Network (RNN) is then used to learn the code transformations performed by developers during PR activities.

We quantitatively and qualitatively evaluate the NMT model. For the quantitative analysis, we assessed its ability in modifying the project's code exactly as done by developers during real PRs. This means that we compare, for the same code components, the output of the manually implemented changes and of the output of the NMT model. The qualitative analysis aims instead at distilling a taxonomy of meaningful code transformations that the model was able to automatically learn from the training data — see Figure 5.1.

The achieved results indicate that, in its best configuration, the NMT model is able to inject the same code transformations that are implemented by developers in PRs in 16-36% of cases, depending on the number of possible solutions that it is required to produce using beam search [158] (*e.g.*, if only the top-ranked solution is picked, the model succeeds in 16% of cases). Moreover, the extracted taxonomy shows that the model is able to learn a rich variety of meaningful code transformations, automatically fixing bugs and refactoring code as humans would do. As explained in Section 5.2, these results have been achieved in a quite narrow context (*i.e.*, we only considered pairs of small/medium methods before/after the implementation of the changes carried by the PR), and this is also one of the reasons why our infrastructure mostly learned bug-fixing and refactoring activities (as opposed to the implementation of new features). However, we believe that our results clearly show the potential of NMT for learning and automating non-trivial code changes and therefore can

pave the way to more research targeting the automation of code changes (*e.g.*, approaches designed to learn and apply refactorings). To foster research in this direction, we make publicly available the complete datasets, source code, tools, and raw data used in our experiments [25].

5.2 Approach

Our approach starts with mining PRs from three large Gerrit repositories (Sec. 5.2.1). From these PRs, we extract the source code *before* and *after* the PRs are merged. We pair the pre-PR and post-PR methods, where each pair serves as an example of a meaningful code change (Sec. 5.2.2). Method pairs are then abstracted, filtered, and organized in datasets (Sec. 5.2.3). Next, we train a RNN Encoder-Decoder to *translate* the version of the code *before* the PR into the version of code *after* the PR, essentially trying to emulate the code change (Sec. 5.2.4). Finally, the output generated by the NMT model is concretized in real source code (Sec. 5.2.5).

5.2.1 Code Reviews Mining

We built a Gerrit crawler to collect the PR data needed to train the NMT model. Given a Gerrit server, the crawler extracts the complete list of projects hosted on it. Indeed, code reviews for multiple projects can be hosted in a single Gerrit server. Then, for each project, the crawler retrieves the list of all PRs submitted for review and having "merged" as the final status (*i.e.*, PRs that had been accepted after the review). We then process each merged PR P using the following steps. First, let us define the set of Java files submitted in P as $F_S = \{F_1, F_2, \ldots, F_n\}$. We ignore non-Java files, since our NMT model only supports Java. For each file in F_S, we use the Gerrit API to retrieve their version before the changes implemented in the PR. The crawler discards new files created in the PR (*i.e.*, not existing before the PR) since we cannot learn any code transformation from them (we need the code before/after the PR to learn changes implemented by developers).

Then, the Gerrit API is used to retrieve the merged version of the files impacted by the PR. The two sets of collected files (*i.e.*, before/after the PR) might not be exactly the same, due to files created/deleted during the code review process (see the next section).

The output of the crawler is, for each PR, the version of the files impacted before (pre-PR) and after (post-PR, merged) the PR. At the end of the mining process we obtain three datasets of PRs: PR_{Ovirt}, $PR_{Android}$, and PR_{Google}.

5.2.2 Code Extraction

Each mined PR is represented as $pr = \{(f_1, \ldots, f_n), (f_1^{'}, \ldots, f_m^{'})\}$, where $f_1, \ldots, f_n$ are the source code files *before* the PR, and $f_1^{'}, \ldots, f_m^{'}$ are code files *after* the PR. As previously explained, the two sets may or may not be the same size, since files could have been added or removed during the PR process. In the first step, we rely on GumTreeDiff [68] to establish the file-to-file mapping, performed using semantic anchors, between pre- and post-PR files and disregarding any file added/removed during the code review process. After this step, each PR is stored in the format $pr = \{(f_1, \ldots, f_k), (f_1^{'}, \ldots, f_k^{'})\}$, where f_i is the file before and $f_i^{'}$ the corresponding version of the file after the PR. Next, each pair of files $(f_i, f_i^{'})$ is again analyzed using GumTreeDiff, which establishes the method-to-method mapping and identifies the AST operations performed between two versions of the same method. We select only the pairs of methods for which the code after the PR has been changed with respect to the code before the PR. Then, each PR is represented as a list of paired methods $pr = \{(m_b, m_a)_1, \ldots, (m_b, m_a)_n\}$, where each pair $(m_b, m_a)_i$ contains the method *before* the PR (m_b) and the method *after* the PR (m_a). We will use these pairs as examples of code changes to train an NMT model to translate m_b in m_a.

We use the method-level granularity for several reasons: (i) methods implement a single functionality and provide enough context for a meaningful code transformation; (ii) file-level code changes are still possible by composing multiple method-level code transformations; (iii) files represent large corpus of text, with potentially many lines of untouched code during the PR, which would hinder our goal to train an NMT model.

Table 5.1: Vocabularies

Dataset	Vocabulary	Abstracted Vocabulary
Google	42,430	373
Android	266,663	429
Ovirt	81,627	351
All	370,519	740

In this chapter we only study code changes which modify existing methods, disregarding code changes that involve the creation or deletion of entire methods/files (see Section 5.5).

5.2.3 Code Abstraction & Filtering

NMT models generate sequences of tokens by computing probability distributions over words. They can become very slow or imprecise when dealing with a large vocabulary comprised of many possible output tokens. This problem has been addressed by artificially limiting the vocabulary size, considering only most common words, assigning special tokens (*e.g.*, `UNK`) to rare words or by learning subword units and splitting the words into constituent tokens [135, 199].

The problem of large vocabularies (a.k.a. open vocabulary) is well known in the Natural Language Processing (NLP) field, where languages such as English or Chinese can have hundreds of thousands of words. This problem is even more pronounced for source code. As a matter of fact, developers are not limited to a finite dictionary of words to represent source code, rather, they can generate a potentially infinite amount of novel identifiers and literals. Table 5.1 shows the number of unique tokens identified in the source code of the three datasets. The vocabulary of the datasets ranges between 42k and 267k, while the combined vocabulary of the three datasets exceeds 370k unique tokens. In comparison, the Oxford English Dictionary contains entries for 171,476 words [147].

In order to allow the training of an NMT model, we need a way to reduce the vocabulary while still retaining semantic information of the source code. We employ an abstraction process which relies on the following observations regarding code changes: (i) several chunks of code might remain untouched; (ii) developers tend to reuse identifiers and literals already

present in the code; (iii) frequent identifiers (*i.e.*, common API calls and variable names) and literals (*e.g.*, 0, 1, "foo") are likely to be introduced in code changes.

We start by computing the top-300 most frequent identifiers (*i.e.*, type, method, and variable names) and literals (*i.e.*, int, double, char, string values) used in the source code for each of the three datasets. This set contains frequent types, API calls, variable names and common literal values (*e.g.*, `0, 1, "\n"`) that we want to keep in our vocabulary.

Subsequently, we abstract the source code of the method pairs by means of a process that replaces identifiers and literals with reusable IDs. The source code of a method is fed to a lexer, built on top of ANTLR [151], which tokenizes the raw code into a stream of tokens. This stream of tokens is then fed into a Java parser, 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 unique identifier and literal is mapped to an ID, having the form of `CATEGORY_#`, where `CATEGORY` represents the type of identifier or literal (*i.e.*, `TYPE`, `METHOD`, `VAR`, `INT`, `FLOAT`, `CHAR`, `STRING`) and # is a numerical ID generated sequentially for each unique type of instance within that category (*e.g.*, the first method will receive `METHOD_0`, the third integer value `INT_2`, *etc.*). These IDs are used in place of identifiers and literals in the abstracted code, while the mapping between IDs and actual identifier/literal values is saved in a map M, which allows us to map back the IDs in the code concretization phase (Section 5.2.5). During the abstraction process, we replace all identifiers/literals with IDs, except for the list of 300 most frequent identifiers and literals, for which we keep the original token value in the corpus.

Given a method pair (m_b, m_a), the method m_b is abstracted first. Then, using the same mapping M generated during the abstraction of m_b, the method m_a is abstracted in such a way that identifiers/literals already available in M will use the same ID, while new identifiers/literals introduced in m_a (and not available in m_b) will receive a new ID. At the end of this process, from the original method pair (m_b, m_a) we obtain the abstracted method pair (am_b, am_a).

We allow IDs to be reused across different method pairs (*e.g.*, the first method name will always receive the ID `METHOD_0`), therefore leading to an overall reduction of the vocabulary size. The third column of Table 5.1 reports the vocabulary size after the abstraction process, which shows a significant reduction in the number of unique tokens in the corpus. In particular, after the abstraction process, the vocabulary contains: (i) Java keywords; (ii) top-300 identifiers/literals; (iii) reusable IDs. It is worth noting that the last row in Table 5.1 (*i.e.*, All) does not represent the cumulative sum, but rather the count of unique tokens when the three dataset corpora are merged.

Having a relatively small vocabulary allows the NMT model to focus on learning patterns of code transformations that are common in different contexts. Moreover, the use of frequent identifiers and literals allows the NMT model to learn typical changes (*e.g.*, `if(i>1)` to `if(i>0)`) and introduce API calls based on other API calls already available in the code.

After the abstraction process, we filter out method pairs from which the NMT model would not be able to learn code transformations that will result in actual source code. To understand the reasoning behind this filtering, it is important to understand the real use case scenarios. When the NMT model receives the source code of the method am_b, it can only perform code transformations that involve: (i) Java keywords; (ii) frequent identifiers/literals; (iii) identifiers and literals already available in m_b. Therefore, we disregard method pairs where m_a contains tokens not listed in the three aforementioned categories, since the model would have to synthesize new identifies or literals not previously seen.

In the future, we plan to increase the number of frequent identifiers and literals used in the vocabulary with the aim of learning code transformations from as many method pairs as possible. We also filter out those method pairs such that $am_b = am_a$, meaning the abstracted code before and after the PR appear the same. We remove these instances since the NMT model would not learn any code transformation.

Next, we partition the method pairs in small and medium pairs, based on their size measured in the number of tokens. In particular, small method pairs are those no longer

Table 5.2: Datasets

Dataset	M_{small}	M_{medium}
Google	2,165	2,286
Android	4,162	3,617
Ovirt	4,456	5,088
All	10,783	10,991

than 50 tokens, while we consider medium pairs those having a length between 50-100 tokens. In this stage, we disregard longer method pairs. We discuss this limitation in Section **??**.

Table 5.2 shows the number of method pairs, after the abstraction and filtering process, for each dataset and the combined one (*i.e.*, All). Each of the four datasets is then randomly partitioned into training (80%), validation (10%), and test (10%) sets. Before doing so, we make sure to remove any duplicate method pairs, to ensure that none of the method pairs in the test set have been seen during the training phase.

5.2.4 Learning Code Transformations

In this section, we describe the NMT models we use to learn code transformations. In particular, we train these models to translate the abstracted code am_b in am_a, effectively simulating the code change performed in the PR by developers.

5.2.4.1 RNN Encoder-Decoder

To build such models, we rely on an RNN Encoder-Decoder architecture with attention mechanism [46, 128, 54], commonly adopted in NMT tasks [106, 171, 59]. As the name suggests, this model consists of two major components: an RNN Encoder, which *encodes* a sequence of tokens $\boldsymbol{x}$ into a vector representation, and an RNN Decoder, which *decodes* the representation into another sequence of tokens $\boldsymbol{y}$. During training, 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 the lengths n and m may differ. In our setting, given the sequence representing the abstract code before the PR $\mathbf{x} = am_b =$

$(x_1, .., x_n)$ and a corresponding target sequence representing the abstract code after the PR $\mathbf{y} = am_a = (y_1, .., y_m)$, the model is trained to learn the conditional distribution: $P(am_a|am_b) = P(y_1, .., y_m|x_1, .., x_n)$, where x_i and y_j are abstracted source tokens: Java keywords, separators, IDs, and frequent identifiers and literals. The Encoder takes as input a sequence $\mathbf{x} = (x_1, .., x_n)$ and produces a sequence of states $\mathbf{h} = (h_1, .., h_n)$. In particular, we adopt a bi-directional RNN Encoder [46], which is formed by a backward and a forward RNN. The RNNs process the sentence both from left-to-right and right-to-left, and are able to create sentence representations taking into account both past and future inputs [54]. 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 token y_i is computed based on: (i) the recurrent state s_i in the Decoder; (ii) the previous $i-1$ tokens $(y_1, .., y_{i-1})$; and (iii) a context vector c_i. This vector c_i, also called attention vector, 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, when predicting the token y_i. Encoder and Decoder are trained jointly by minimizing the negative log likelihood of the target tokens, using stochastic gradient descent.

5.2.4.2 Beam Search Decoding

For each method pair (am_b, am_a) the model is trained to translate am_b solely into the corresponding am_a. However, during testing, we would like to obtain *multiple* possible translations. Precisely, given a piece of source code m as input to the model, we would like to obtain k possible translations of m. To this aim, we employ a decoding strategy called a Beam Search used in previous applications of DL [158]. The major intuition behind a 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 until 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:

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

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.

5.2.4.3 Hyperparameter Search

We performed hyperparameter search by testing ten configurations of the encoder-decoder architecture. The configurations tested different combinations of RNN Cells (LSTM [96] and GRU [59]), 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 the 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/settings are listed in our online appendix [25].

5.2.5 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.

5.3 Experimental Design

The *goal* of this study is to empirically assess whether NMT can be used to learn a diverse and meaningful set of code changes. The *context* consists of a dataset of PRs and aims at answering two research questions (RQs).

5.3.1 RQ1: Can Neural Machine Translation be employed to learn meaningful code changes?

We aim to empirically assess whether NMT is a viable approach to learn transformations of the code, as performed by developers in PRs. To this end, we use the eight datasets of method pairs listed in Table 5.2. Given a dataset, we train different configurations of the Encoder-Decoder models on the training set, then use the validation set to select the best performing configuration of the model. We then evaluate the validity of the model with the unseen instances of the test set. In total, we experiment with eight different models, one for each dataset in Table 5.2 (*i.e.*, one model trained, configured, and evaluated on the Google dataset of small methods, one on the Google dataset of medium methods, *etc.*).

The evaluation is performed by the following methodology. Let M be a trained model and T be the test set of dataset D, we evaluate the model M for each $(am_b, am_a) \in T$. Specifically, we feed the pre-PR abstract code am_b to the model M, performing inference with Beam Search Decoding for a given beam size k. The model will generate k different potential code transformations $CT = \{ct^1, \ldots, ct^k\}$. We say that the model successfully predicted a code transformation if there exists a $ct^i \in CT$ such that $ct^i = am_a$ (*i.e.*, the abstract code generated by developers after the merging of the PR). We report the raw count and percentage of successfully predicted code changes in the test set, with $k = 1, 5, 10$. In other words, given a source code method that the model has never seen before, we evaluate the model's ability to correctly predict the code transformation that a developer performed by allowing the model to generate its best guess (*i.e.*, $k = 1$) or the top-5 and top-10 best guesses. It should be noted that while we count only perfect

Table 5.3: Perfect Predictions

Dataset	Beam	M_{small}	M_{medium}
Google	1	10 (4.62%)	7 (3.07%)
	5	17 (7.87%)	13 (5.70%)
	10	20 (9.25%)	17 (7.45%)
Android	1	40 (9.61%)	51 (14.12%)
	5	71 (17.06%)	73 (20.22%)
	10	79 (18.99%)	76 (21.05%)
Ovirt	1	55 (12.35%)	60 (11.78%)
	5	93 (20.89%)	90 (17.68%)
	10	113 (25.39%)	102 (20.03%)
All	1	228 (21.16%)	178 (16.21%)
	5	349 (32.40%)	306 (27.86%)
	10	388 (36.02%)	334 (30.41%)

predictions, there are many other (slightly different) transformations that can still be viable and useful for developers. However, we discount these less-than-perfect predictions since it is not possible to automatically categorize those as viable and non-viable.

5.3.2 RQ2: What types of meaningful code changes can be performed by the model?

In this RQ we aim to qualitatively assess the types of code changes that the NMT model is able to generate. To this goal, we focus only on the successfully predicted code transformations generated by the model trained on the *All* dataset, considering both small and medium sized methods.

One of the authors manually investigated all the successfully predicted code transformations and described the code changes. Subsequently, a second author discussed and validated the described code changes. Finally, the five authors together defined – and iteratively refined – a taxonomy of code transformations successfully performed by the NMT model.

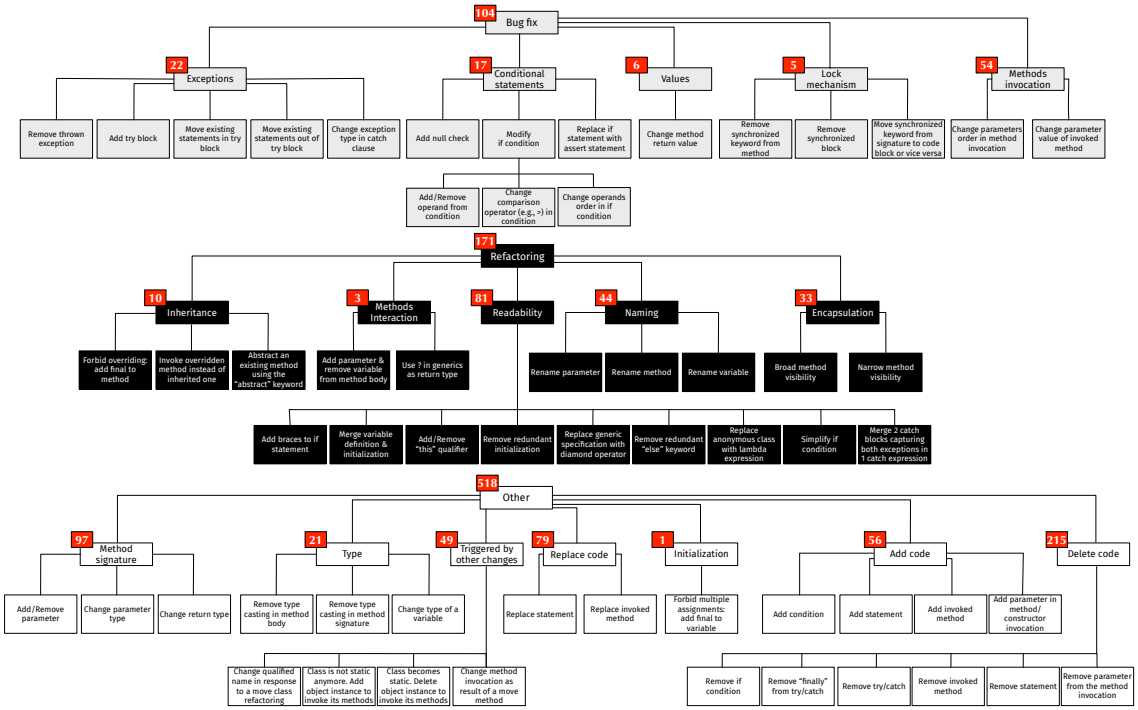

Figure 5.1: Taxonomy of code transformations learned by the NMT model

5.4 Results

5.4.1 RQ1: Can Neural Machine Translation be employed to learn meaningful code changes?

Table 5.3 reports the perfect predictions (*i.e.*, successfully predicted code transformations) by the NMT models, in terms of raw numbers and percentages of the test sets. When we allow the models to generate only a single translation (*i.e.*, beam = 1), they are able to predict the same code transformation performed by the developers in 3% up to 21% of the cases. It is worth noting how the model trained on the combined datasets (*i.e.*, All) is able to outperform all the other single-dataset model, achieving impressive results even with a single guess (21.16% for small and 16.21% for medium methods). This result shows that NMT models are able to learn code transformations from a heterogeneous set of examples belonging to different datasets. Moreover, this also provides preliminary evidence that transfer learning would be possible for such models.

On the other end of the spectrum, the poor performance of the models trained on Google's dataset could be explained by the limited amount of training data (see Table 5.2) with respect to the other datasets.

When we allow the same models to generate multiple translations of the code (*i.e.*, 5 and 10), we observe a significant increase in perfect predictions across all models. On average, 1 out of 3 code transformations can be generated and perfectly predicted by the NMT model trained on the combined dataset.

Summary for RQ$_1$. NMT models are able to learn meaningful code changes and perfectly predict code transformations in up to 21% of the cases when only one translation is generated, and up to 36% when 10 possible guesses are generated.

5.4.2 RQ2: What types of meaningful code changes can be performed by the model?

Here we focus on the 722 (388+334) perfect predictions generated by the model trained on the whole dataset, *i.e.*, All, with beam size equals 10. These perfect predictions were the results of 216 unique types of AST operations, as detected by GumTreeDiff, that the model was able to emulate. The complete list is available in our replication package [25].

Figure 5.1 shows the taxonomy of code transformations that we derived by manually analyzing the 722 perfect predictions. Note that a single perfect prediction can include multiple types of changes falling into different categories of our taxonomy (*e.g.*, a refactoring and a bug fix implemented in the same code transformation). For this reason, the sum of the classified changes in Figure 5.1 is 793. The taxonomy is composed of three sub-trees, grouping code transformations related to bug fixing, refactoring, and "other" types of changes. The latter includes code transformations that the model correctly performed (*i.e.*, those replicating what was actually done by developers during the reviewed PRs) but for which we were unable to understand the rationale behind the code transformation (*i.e.*, why it was performed). We preferred to adopt a conservative approach

and categorize transformations into "refactoring" and "bug-fix" sub-trees only when we can confidently link to these types of activities. Also, for 27 transformations, the authors did not agree on the type of code change and, for this reason, we excluded them from our taxonomy (that, thus, is related to 695 perfect predictions).

Here, we qualitatively discuss interesting examples (indicated using the 🌿 icon) of code transformations belonging to our taxonomy. We do not report examples for all possible categories of changes learned by the model due to lack of space. Yet, the complete set of perfect predictions and their classification is available in our replication package [25].

5.4.3 Refactoring

We grouped in the refactoring sub-tree, all code transformations that modify the internal structure of the system by improving one or more of its non-functional attributes (*e.g.*, readability) without changing the system's external behavior. We categorized the code transformations into five sub-categories.

5.4.3.1 Inheritance

Refactorings that impact how the inheritance mechanism is used in the code. We found three types of refactorings related to inheritance: (i) forbid method overriding by adding the `final` keyword to the method declaration; (ii) invoke overriding method instead of overridden by removing the `super` keyword to the method invocation; and (iii) making a method abstract through the `abstract` keyword and deleting the method body.

🌿 **Existing method declared as final [3]**. In the `DirectByteBuffer` class of Android, the NMT model added to the signature of the `getLong(int)` method the `final` keyword. As stated by the developer implementing the PR: "*DirectByteBuffer cannot be final, but we can declare most methods final to make it easier to reason about*".

🌿 **Removed unnecessary "super" specifier [28].** A PR in the Ovirt core subsystem was performed to clean up the class `RandomUtil`, that extends Java class `java.util.Random`.

The `nextShort`() method implemented in the refactored class was invoking `nextInt`() of the base class through the use of the `super` java specifier. However, such a specifier was redundant because `nextInt`() was not overridden in `RandomUtil`. Thus, it was removed by the developer: "*Using this modifier has no meaning in the context that was removed*".

Existing method converted to abstract [1].

```
1  float getFloatUnchecked(int index) {
2    throw new UnsupportedOperationException();
3  }
4
5  abstract float getFloatUnchecked(int index);
```

The above code listing shows the code taken as input by the NMT model (top part, pre-PR) and produced as output (bottom, post-PR). The code transformation replicates the changes implemented by a developer in a PR, converting the `getFloatUnchecked` method into an abstract method, deleting its body. The rationale for this change is explained by the developer who implemented this change: The method `getFloatUnchecked` is overridden in all child classes of the abstract class implementing it and, thus, "*there is no need for the abstract base class to carry an implementation that throws* `UnsupportedOperationException`". The developer also mentions alternative solutions, such as moving this and similar methods into an interface, but concludes saying that the effort would be much higher. This case is interesting for at least two reasons. First, our model was able to learn a combination of code transformations needed to replicate the PR implemented by the developer (*i.e.*, add the `abstract` keyword *and* delete the method body). Second, it shows the rich availability of information about the "rationale" for the implemented changes available in code review repositories. This could be exploited in the future to not only learn the code transformation, but also to justify it by automatically deriving the rationale from the developers' discussion.

5.4.3.2 Methods Interaction

These refactorings impact the way in which methods of the system interact, and include (i) *add parameter* refactoring (*i.e.*, a value previously computed in the method body is now passed as parameter to it), and (ii) broadening the return type of a method by using the Java wildcard (?) symbol.

Method returns a broader generic type [20].

```
1 <I> RestModifyView<P,I> post(P parent) throws [...];
2
3 RestModifyView<P,?> post(P parent) throws [...];
```

The code listing shows a change implemented in a PR done on the "Google" Gerrit repository and correctly replicated by the NMT model. The `post` method declaration was refactored to return a broader type and improve the usage of generics. As explained by the developer, this also allows to avoid the 'unchecked' warnings from the five implementations of the `post` method present in the system, thus simplifying the code.

5.4.3.3 Naming

This category groups refactorings related to the renaming of methods, parameters, and variables. This is usually done to improve the expressiveness of identifiers and to better adhere to the coding style guidelines. Indeed, good identifiers improve readability, understandability and maintainability of source code [36, 97].

Rename method [29]. One example of correctly learned rename method, is the one fixing a typo from the `OnSucess` method in the Ovirt system [29]. In this case, the developer (and the NMT model) both suggested to rename the method in `OnSuccess`.

Rename parameter [12]. A second example of renaming, is the renamed parameter proposed for the `endTrace(JMethod type)` method in a PR impacting the `AbstractTracerBrush` class in the Android repository [12]. The developer here renamed several parameters "for

clarity" and, in this case, renamed the `type` parameter into `method`, to make it more descriptive and better reflect its aim.

5.4.3.4 Encapsulation

We found refactorings aimed at broadening and narrowing the visibility of methods (see Figure 5.1). This can be done by modifying the access modifiers (*e.g.*, changing a public method to a private one).

Broadening [5] and narrowing [21] method visibility. An example of a method, for which our model recommended to broaden its visibility from `private` to `public`, is the `of` method from the `Key` Android class [5]. This change was done in a PR to allow the usage of the method from outside the class, since the developer needed it to implement a new feature.

The visibility was instead narrowed from `public` to `private` in the context of a refactoring performed by a developer to make "*more methods private*" [21]. This change impacted the `CurrentUser.getUser()` method from the Google repository, and the rationale for this change correctly replicated by the NMT model was that the `getUser()` method was only used in one location in the system outside of its class. However, in that location the value of "*the user is already known*", thus do not really requiring the invocation of `getUser()`.

5.4.3.5 Readability

Readable code is easier to understand and maintain [163]. We found several types of code transformations learned by the model and targeting the improvement of code readability. This includes: (i) braces added to `if` statements with the only goal of clearly delimiting their scope; (ii) the merging of two statements defining (*e.g.*, `String address;`) and initializing (*e.g.*, `address = getAddess();`) a variable into a single statement doing both (*e.g.*, `String address = getAddess();`); (iii) the addition/removal of the `this` qualifier, to match the project's coding standards; (iv) reducing the verbosity of a generic declaration by

using the Java diamond operator (*e.g.*, `Map < String, List < String >> mapping = new HashMap < String, List < String >> ()` becomes `Map < String, List < String >> mapping = new HashMap <>`); (v) remove redundant `else` keywords from `if` statements (*i.e.*, when the code delimited by the `else` statement would be executed in any case); (vi) refactoring anonymous classes implementing one method to lambda expressions, to make the code more readable [26]; (vii) simplifying boolean expressions (*e.g.*, `if(x == true)` becomes `if(x)`, where `x` is a boolean variable); and (viii) merging two catch blocks capturing different exceptions into one catch block capturing both exceptions using the `or` operator [7].

Anonymous class replaced with lambda expression [26].

```
1  public boolean isDiskExist([...]) {
2    return execute(new java.util.concurrent.Callable<java.lang.Boolean>()
         {
3      @java.lang.Override
4      public java.lang.Boolean call() { try {[...]} } }); }
5
6  public boolean isDiskExist([...]) {
7    return execute(() -> { try {[...]} }); }
```

In the above code listing, the NMT model automatically replaces an anonymous class (top part, pre-PR) with a lambda expression (bottom part, post-PR), replicating changes made by Ovirt's developers during the transitions of the code through Java 8. The new syntax is more compact and readable.

Merging catch blocks capturing different exceptions [7].

```
1  public static Integer getInteger(String nm, Integer val) {
2    [...]
3    try {[...]}
4    catch (IllegalArgumentException e) { }
5    catch (NullPointerException e) { }
6  }
7
```

```
8  public static Integer getInteger(String nm, Integer val) {
9    [...]
10   try {[...]}
11   catch (IllegalArgumentException | NullPointerException e) { }
12 }
```

As part of a PR implementing several changes, the two `catch` blocks of the `getInteger` method were merged by the developer into a single `catch` block (see the code above). The NMT model was able to replicate such a code transformation that is only meaningful when an exception is caught and the resulting code that is executed is the same for both instances of the exception (as in this case). This code change, while simple from a developer's perspective, is not trivial to learn due to the several transformations to implement (*i.e.*, removal of the two `catch` blocks and implementation of a new `catch` block using the | or operator) and to the "pre-condition" to check (*i.e.*, the same behavior implemented in the catch blocks).

5.4.4 Bug Fix

Changes in the "bug fix" subtree (see Figure 5.1) include changes implemented with the goal of fixing a specific bug which has been introduced in the past. The learned code transformations are organized here into five sub-categories, grouping changes related to bug fixes that deal with (i) exception handling, (ii) the addition/modification of conditional statements, (iii) changes in the value returned by a method, (iv) the handling of lock mechanisms, and (v) wrong method invocations.

5.4.4.1 Exception

This category of changes is further specialized into several subcategories (see Figure 5.1) including (i) the addition/delation of thrown exceptions; (ii) the addition of `try` − `catch`/`finally` blocks [2]; (iii) narrowing or broadening the scope of the `try` block by moving the exist-

ing statements inside/outside the block [9]; (iv) changing the exception type in the catch clause to a narrower type (*e.g.*, replacing `Throwable` with `RuntimeException`).

Add try-catch block [2].

```
public void test_getPort() throws IOException {
  DatagramSocket theSocket = new DatagramSocket();
  [...]
}

public void test_getPort() throws IOException {
  try (DatagramSocket theSocket = new DatagramSocket()) {
  [...]
  }
}
```

The above code from the Android repository, shows the change implemented in a PR aimed at fixing "*resource leakages in tests*". The transformation performed by the NMT model wrapped the creation and usage of a `DatagramSocket` object into a `try − with − resources` block. This way `theSocket.close()` will be automatically invoked (or an exception will be thrown), thus avoiding resource leakage.

Narrowed the scope of try block [9].

```
public void testGet_NullPointerException() {
  try {
    ConcurrentHashMap c = new ConcurrentHashMap(5);
    c.get(null);
    shouldThrow();
  } catch (java.lang.NullPointerException success) {}
}

public void testGet_NullPointerException() {
  ConcurrentHashMap c = new ConcurrentHashMap(5);
  try {
    c.get(null);
```

```
13      shouldThrow();
14    } catch (java.lang.NullPointerException success) {}
15  }
```

Another change replicated by the NMT model and impacting the Andorid test suite is the code transformation depicted above and moving the `ConcurrentHashMap` object instantiation outside of the `try` block. The reason for this change is the following. The involved test method is supposed to throw a `NullPointerException` in case `c.get(null)` is invoked. Yet, the test method would have also passed if the exception was thrown during the `c` instantiation. For this reason, the developer moved the object creation out of the `try` block.

5.4.4.2 Conditional statements

Several bugs can be fixed in conditional statements verifying that certain preconditions are met before specific actions are performed (*e.g.*, verifying that an object is not null before invoking one of its methods).

Added null check [4].

```
1  public void run() {
2    mCallback.onConnectionStateChange(BluetoothGatt.this, GATT_FAILURE,
3    BluetoothProfile.STATE_DISCONNECTED);
4  }
5
6  public void run() {
7    if (mCallback != null) {
8      mCallback.onConnectionStateChange(BluetoothGatt.this, GATT_FAILURE,
9      BluetoothProfile.STATE_DISCONNECTED);
10   }
11 }
```

The code listing shows the changes implemented in an Android PR to "*fix a NullPointerException when accessing mCallback in BluetoothGatt*". The addition of the `if` statement

implementing the null check allows the NMT model to fix the bug exactly as the developer did.

Change comparison operand [6].

```
1  public void reset(int i) {
2    if ((i < 0) || (i >= mLen)) { [...] }
3  }
4
5  public void reset(int i) {
6    if ((i < 0) || (i > mLen)) { [...] }
7  }
```

A second example of a bug successfully fixed by the NMT model working on the conditional statements, impacted the API of the `FieldPacker` class. As explained by the developer, the PR contributed "*a fix to the FieldPacker.reset() API, which was not allowing the FieldPacker to ever point to the final entry in its buffer*". This was done by changing the >= operand to > as shown in the code reported above.

5.4.4.3 Values

The only type of change we observed in this category is the change of methods' return value to fix a bug. This includes simple cases in which a `boolean` return value was changes from `false` to `true` (see *e.g.*, [13]), as well as less obvious code transformations in which a constant return value was replaced with a field storing the current return value, *e.g.*, `return "refs/my/config"`; converted into `return ref;`, where `ref` is a variable initialized in the constructor [22].

5.4.4.4 Lock mechanism

These code changes are all related to the usage of the `synchronized` Java keyword in different parts of the code. These include its removal from a code block [11], from a method signature [10], and moving the keyword from the method signature to a code block or *vice*

versa [8]. We do not discuss these code transformations due to their simplicity and lack of space.

5.4.4.5 Methods invocation

These category groups code transformations fixing bugs by changing the order or value of parameters in method invocations.

Flipped parameters in assertEquals [27].

```
1 public void testConvertMBToBytes() {
2   [...]
3   org.junit.Assert.assertEquals(bytes, 3145728);
4 }
5
6 public void testConvertMBToBytes() {
7   [...]
8   org.junit.Assert.assertEquals(3145728, bytes);
```

In this example the developer fixed a bug in the test suite by flipping the order in which the parameters are passed to the `assertEquals` method. In particular, while the assert method was expecting the pairs of parameters (`long expected`, `long actual`), test was passing the actual value first, thus invalidating the test. The fix, automatically applied by the NMT model, swaps the arguments of the `assertEquals`.

5.4.5 Other

As previously said, we assigned to the 'Other' subtree those code transformations for which we were unable to clearly identify the motivation/reason. This subtree includes changes related to: (i) the method signature (added/removed/changed parameter or return type); (ii) types (removed type casting in method body or its signature, changed variable type); (iii) variable initialization; (iv) replaced statement/invoked method; (v) added code (condition, statement, invoked method, parameter); (vi) deleted code (`if` condition, `finally`

block, `try − catch` block, invoked method, statement); (vii) changes triggered by the other changes (*e.g.*, static method call replaced with an instance method call or *vice versa* — see Figure 5.1). Note that, while we did not assign a specific "meaning" to these changes, due to a lack of domain knowledge of the involved systems, these are still perfect predictions that the NMT model performed. This means the code changes are identical to the ones implemented by developers in the PR. While we do not show examples due to lack of space, all classified code transformations are available in our replication package [25].

Summary for RQ$_2$. Our results show the great potential of NMT for learning meaningful code changes. Indeed, the NMT model was able to learn and automatically apply a wide variety of code changes, mostly related to refactoring and bug-fixing activities. The fact that we did not find other types of changes, such as new feature implementation, might be due to the narrow context in which we applied our models (*i.e.*, methods of limited size), as well as to the fact that new features implemented in different classes and systems rarely exhibit recurring patterns (*i.e.*, recurring types of code changes) that the model can learn. More research is needed to make this further step ahead.

5.5 Threats to Validity

Construct validity. We collected code components before and after pull requests through a crawler relying on the Gerrit API. The crawler has been extensively tested, and the manual analysis of the extracted pairs performed to define the taxonomy in Figure 5.1 confirmed the correctness of the collected data.

Internal validity. The performance of the NMT model might be influenced by the hyperparameter configuration we adopted. To ensure replicability, we explain in Section 5.2 how hyperparameter search has been performed.

We identified through the manual analysis the types of code transformations learned by the model. To mitigate subjectivity bias in such a process, the taxonomy definition has been done by one of the authors, double checked by a second author, and finally, the

resulting taxonomy has been discussed among all authors to spot possible issues. Moreover, in case of doubts, the code transformation was categorized in the "other" subtree, in which we only observed the type of code change implemented, without conjecturing about the goal of the transformation. However, as in any manual process, errors are possible, and we cannot exclude the presence of misclassified code transformations in our taxonomy.

External validity. We experimented with the NMT model on data related to Java programs only. However, the learning process is language-independent and the whole infrastructure can be instantiated for different programming languages by replacing the lexer, parser and AST differencing tools.

We only focused on methods having no more than 100 tokens. This is justified by the fact that we observe a higher density of method pairs with sizes less than 100 tokens in our dataset. The distribution also shows a long tail of large methods, which could be problematic when training a NMT model. Distribution and data can be accessed in our replication package [25]. Also, we only focus on learning code transformations of existing methods rather than the creation of new methods since these latter are (i) complex code changes that involve a higher level of understanding of the software system in its entirety; and (ii) not well-suited for NMT models since the translation would go from/to empty methods.

Finally, pull request data from three Gerrit repositories were used. While these repositories include hundreds of individual projects (thus ensuring a good external validity of our findings) our results might not generalize to other projects/languages.

5.6 Related Work

Deep Learning (DL) has recently become a useful tool to study different facets of software engineering. The unique representations allow for features to be discovered by the model rather than manual derivation. Due to the power of these representations, many works have applied these models to solve SE problems [76][34][66][61][123][90][160][88]. However,

to the best of our knowledge, this is the first work that uses DL techniques to learn and create a taxonomy from a variety of code transformations taken from developers' PRs.

White *et al.* uses representation learning via a recursive autoencoder for the task of clone detection [196]. Each piece of code is represented as a stream of identifiers and literals, which they use as input to their DL model. Using a similar encoding, Tufano *et al.* encodes methods into four different representations, then the DL model evaluates how similar two pieces of code are based on their multiple representations [180]. Another recent work by Tufano *et al.* applies NMT to bug-fixing patches the wild [181]. This work applies a similar approach, but rather than learning code transformations they attempt to learn bug-fixing commits to generate patches. These works are related to ours, since we use a similar code representation as input to the DL model, yet, we apply this methodology to learn as many code transformations as possible.

White *et al.* also compare DL models with natural language processing models for the task of code suggestion. They show that DL models make code suggestions based upon contextual features learned by the model rather than the predictive power of the past n tokens [197]. Further expanding upon the powerful, predictive capabilities of these models, Dam *et al.* presents DeepSoft, which is a DL-based architecture used for modeling software, code generation and software risk prediction [62].

DL has also been applied to the areas of bug triaging and localization. Lam *et al.* makes use of DL models and information retrieval to localize buggy files after a bug report is submitted. They use a revised Vector Space Model to create a representation the DL model can use to relate terms in a bug report to source code tokens [114]. Likewise, to reduce the effort of bug triaging, Lee *et al.* applies a CNN to industrial software in order to properly triage bugs. This approach uses word2vec to embed a summary and a description which the CNN then assigns to a developer [122]. Related to software bugs, Wang *et al.* uses a Deep Belief Network (DBN) to learn semantic features from token vectors taken from a programs' ASTs. The network then predicts if the commit will be defective [190].

Many DL usages aim to help developers with tasks outside of writing code. Choetkiertikul *et al.* proposes a DL architecture of long short-term memory and recurring highway network that aims to predict the effort estimation of a coding task [60]. Another aid for developers is the ability to summarize a given segment of source code. To this point Allamanis *et al.* uses an Attentional Neural Network (ANN) with a convoluation layer in order to summarize pieces of source code into short, functional descriptions [40]. Guo *et al.* develops a DL approach using RNNs and word embeddings to learn the sentence semantics of requirement artifacts, which helps to create traceability links in software projects [86]. The last example of DL implementations that aid developers in the software development process is an approach developed by Gu *et al.* that helps to locate source code. This implementation uses NNs and natural language to embed code snippets with natural language descriptions into a high-dimensional vector space, helping developers locate source code based on natural language queries [82].

DL-based approaches have also been applied to more coding related tasks, one such task is accurate method and class naming. Allamanis *et al.* uses a log-bilinear neural network to understand the context of a method or class and recommends a representative name that has not appeared in the training corpus [35]. Also helping with correct coding practices, Gu *et al.* uses an RNN encoder-decoder model to generate a series of correct API usages in source code based upon natural language queries. The learned semantics allow the model to associate natural language queries with a sequence of API usages [84].

Recently we have seen DL infiltrate the mobile SE realm. Moran *et al.* uses a DL-based approach to automatically generate GUIs for mobile apps. In this approach, a deep CNN is used to help classify GUI components which can later be used to generate a mock GUI for a specific app [137].

Although DL approaches are prevalent in SE, this work is the first to apply DL to empirically evaluate the capability to learn code changes from developer PRs. The previous work has shown that DL approaches can yield meaningful results given enough quality

training data. Thus, we specifically apply NMT to automatically learn a variety of code transformations, from real pull requests, and create a meaningful taxonomy.

5.7 Conclusion

In this chapter we quantitatively and qualitatively investigated the ability of Neural Machine Translation (NMT) models to learn how to automatically apply code transformations. We first mine a dataset of complete and meaningful code changes performed by developers in merged pull requests, extracted from three large Gerrit repositories. Then, we train NMT models to translate pre-PR code into post-PR code, effectively learning code transformations as performed by developers.

Our empirical analysis shows that NMT models are capable to learn code changes and perfectly predict code transformations in up to 21% of the cases when only a single translation is generated, and up to 36% when 10 possible guesses are generated. The results also highlight the ability of the models to learn from a heterogeneous set of PRs belonging to different dataset, indicating the possibility of transfer learning across different projects and domains.

The performed qualitative analysis also highlighted the ability of the NMT models to learn a wide variety of meaningful code transformations, paving the way to further research in this field targeting the automatic learning and application of non-trivial code changes, such as refactoring operations. In that sense, we hope that the public availability of the source code of our infrastructure and of all the data and tools used in our study [25], can help in fostering research in this field.

Chapter 6

Conclusions & Future Research

In this dissertation, we have presented a novel approach to learn code transformations via Neural Machine Translation in the context of three major software engineering tasks: (i) Mutation Testing; (ii) Automated Program Repair; and (iii) Learning Code Changes. The overarching motivation of this work has been to build intelligent systems that can learn from real world data with the long-lasting goal of automating different software developers' activities.

We start by mining real world examples of code transformations performed by developers in the wild, from publicly available repositories on GitHub and code review systems such as Gerrit. From this data, we perform fine grained AST differencing and extract method-level transformation pairs that represents the data points we will use to train our Neural Machine Translation models to learn from examples. Next, we perform code abstraction, a process we devised, which transforms the source code into an abstract representation with limited vocabulary size that also retains syntactical and semantical information. We train an RNN Encoder-Decoder models with these transformation pairs by learning to translate the abstract code before the change into the abstract code after the change. We thoroughly evaluate the models across different dimensions: quality of the translation, the percentage of successful changes, the syntactic correctness of the proposed sentences, the percentage of emulated AST operations, and the inference time of the models.

In the first project we use bug fixes to train a model that learns how to mutate source code. The model is trained to translate the fixed code into the original buggy code, therefore, learning to introduce bugs. The results show that the models are able to generate syntactically correct mutants that resemble real bugs. As future work, we plan to deploy a full-fledged mutation tool: DeepMutation. Moreover, we plan to provide more empirical evidence on the types of mutations generated by the model and possibly novel types of mutants not available in the literature.

In the second project, we instantiate this general idea in the Automated Program Repair domain. We train NMT models to translate buggy code into fixed code. We rely on beam search to generate many different translations for the same buggy code, to have a large set of candidate patches. The results show that the model can successfully fix (exactly as the developer originally did) thousands of unique bug-fixes that were never seen in the training set. As future research, we are already involved in extending this idea by providing the model with even more context (not only the buggy method, but also the buggy class) and integrating this model into a framework that also tests the candidate patches against the available test suite. SequenceR [58] is the result of this effort and collaboration with other researchers in the APR community.

In the last chapter of this dissertation, we push our learning infrastructure to learn not only mutations and bug-fixes, but many different types of code changes. With this aim, we mine meaningful code changes, from code review systems, that have undergone a code review process and have been accepted and merged in the master branch. The results demonstrate that the models are able to successfully replicate many different types of changes. These types of changes have been manually classified in a taxonomy, that shows that the models are able to perform refactoring operations, bug-fixes, and other changes that modify the behavior of methods. As future work, we plan to integrate these models in DevOps pipelines and code review systems.

Bibliography

[1] Android: Abstract Method. https://android-review.googlesource.com/c/platform/libcore/+/675863.

[2] Android: Add Catch Block. https://android-review.googlesource.com/c/platform/libcore/+/283122.

[3] Android: Add Final. https://android-review.googlesource.com/c/platform/libcore/+/321410/1/.

[4] Android: Added Null Check. https://android-review.googlesource.com/c/platform/frameworks/base/+/382232.

[5] Android: Broadening Visibility. https://android-review.googlesource.com/c/platform/tools/base/+/110627/6/.

[6] Android: Change Operand. https://android-review.googlesource.com/c/platform/frameworks/base/+/98463/2/.

[7] Android: Merging Catch Blocks. https://android-review.googlesource.com/c/platform/libcore/+/244295/4/.

[8] Android: Move Synchronization. https://android-review.googlesource.com/c/platform/libcore/+/40261/2/.

[9] Android: Narrow Catch Block. https://android-review.googlesource.com/c/platform/libcore/+/148551.

[10] Android: Remove Synchronized From Signature. https://android-review.googlesource.com/c/platform/frameworks/base/+/114871/2/.

[11] Android: Remove Synchronized. https://android-review.googlesource.com/c/platform/frameworks/base/+/143346.

[12] Android: Rename Parameter. https://android-review.googlesource.com/c/toolchain/jack/+/264513/2/.

[13] Android: Return Value. https://android-review.googlesource.com/c/platform/tools/base/+/155460/6/.

[14] Bugzilla. https://www.bugzilla.org/.

[15] Gerrit - Android. https://android-review.googlesource.com/ (last access: 18/08/2018).

[16] Gerrit - Google Source. https://gerrit-review.googlesource.com/ (last access: 18/08/2018).

[17] Gerrit - Ovirt. https://gerrit.ovirt.org/ (last access: 18/08/2018).

[18] Gerrit. https://www.gerritcodereview.com (last access: 11/08/2018).

[19] Git. https://git-scm.com/.

[20] Google: Broader Generic Type. https://gerrit-review.googlesource.com/c/gerrit/+/127039.

[21] Google: Narrowing Visibility. https://gerrit-review.googlesource.com/c/gerrit/+/99660/4/.

[22] Google: Return Value. https://gerrit-review.googlesource.com/c/gerrit/+/139770.

[23] Google Translate. https://translate.google.com/.

[24] Jira. https://www.atlassian.com/software/jira.

[25] On learning meaningful code changes via neural machine translation Replication Package https://sites.google.com/view/learning-codechanges.

[26] Ovirt: Anonymous Class To Lambda. https://gerrit.ovirt.org/#/c/50859/.

[27] Ovirt: Flipped Parameters. https://gerrit.ovirt.org/#/c/63570/.

[28] Ovirt: Redundant Super. https://gerrit.ovirt.org/#/c/45678/.

[29] Ovirt: Rename Method. https://gerrit.ovirt.org/#/c/14147/.

[30] Svn. https://subversion.apache.org/.

[31] Jester - the junit test tester. http://jester.sourceforge.net, 2000.

[32] Pit. http://pitest.org/, 2010.

[33] Abdulkareem Alali, Huzefa H. Kagdi, and Jonathan I. Maletic. 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*, pages 182–191, 2008.

[34] Carol V. Alexandru. Guided code synthesis using deep neural networks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 1068–1070, New York, NY, USA, 2016. ACM.

[35] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, New York, NY, USA, 2015. ACM.

[36] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, New York, NY, USA, 2015. ACM.

[37] Miltiadis Allamanis, Earl T. Barr, René Just, and Charles A. Sutton. Tailored mutants fit bugs better. *CoRR*, abs/1611.02516, 2016.

[38] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *CoRR*, abs/1711.00740, 2017.

[39] Miltiadis Allamanis, Pankajan Chanthirasegaran, Pushmeet Kohli, and Charles Sutton. Learning continuous semantic representations of symbolic expressions. *arXiv preprint arXiv:1611.01423*, 2016.

[40] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. A convolutional attention network for extreme summarization of source code. *CoRR*, abs/1602.03001, 2016.

[41] James H. Andrews, Lionel C. Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 402–411, 2005.

[42] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Software Eng.*, 32(8):608–624, 2006.

[43] Anonymized. Online Appendix. `https://sites.google.com/view/learning-fixes`, 2018.

[44] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. 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*, page 23, 2008.

[45] Andrea Arcuri and Xin Yao. 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*, pages 162–168, 2008.

[46] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[47] Junwei Bao, Nan Duan, Ming Zhou, and Tiejun Zhao. Knowledge-based question answering as machine translation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 967–976, 2014.

[48] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 306–317, New York, NY, USA, 2014. ACM.

[49] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering*, 40(7):671–694, July 2014.

[50] Jacqueline Berkman. Machine learning vs. deep learning, August 2018. [Online; posted 22-August-2017].

[51] Carlos Bernal-Cardenas, Kevin Moran, Michele Tufano, Zichang Liu, Linyong Nan, Zhehan Shi, and Denys Poshyvanyk. Guigle: A gui search engine for android apps. *arXiv preprint arXiv:1901.00891*, 2019.

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

[53] Lionel C. Briand, Massimiliano Di Penta, and Yvan Labiche. Assessing and improving state-based class testing: A series of experiments. *IEEE Trans. Software Eng.*, 30(11):770–793, 2004.

[54] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc V. Le. Massive exploration of neural machine translation architectures. *CoRR*, abs/1703.03906, 2017.

[55] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 511–522, New York, NY, USA, 2017. ACM.

[56] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino, and Mauro Pezzè. Automatic recovery from runtime failures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 782–791, Piscataway, NJ, USA, 2013. IEEE Press.

[57] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 597–608, Piscataway, NJ, USA, 2017. IEEE Press.

[58] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *CoRR*, abs/1901.01808, 2019.

[59] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[60] M. Choetkiertikul, H. K. Dam, T. Tran, T. T. M. Pham, A. Ghose, and T. Menzies. A deep learning model for estimating story points. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[61] C. S. Corley, K. Damevski, and N. A. Kraft. Exploring the use of deep learning for feature location. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 556–560, Sept 2015.

[62] Hoa Khanh Dam, Truyen Tran, John Grundy, and Aditya Ghose. Deepsoft: A vision for a deep model of software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 944–947, New York, NY, USA, 2016. ACM.

[63] Muriel Daran and Pascale Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis, ISSTA 1996, San Diego, CA, USA, January 8-10, 1996*, pages 158–171, 1996.

[64] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[65] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt. Towards mutation analysis of android apps. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–10, April 2015.

[66] J. Deshmukh, A. K. M, S. Podder, S. Sengupta, and N. Dubash. Towards accurate duplicate bug retrieval using deep learning techniques. In *2017 IEEE Interna-*

tional Conference on Software Maintenance and Evolution (ICSME), pages 115–124, Sept 2017.

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

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

[69] Michael Fischer, Martin Pinzger, and Harald C. Gall. 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*, page 23, 2003.

[70] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2018.

[71] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 416–419, 2011.

[72] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Trans. Software Eng.*, 39(2):276–291, 2013.

[73] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 147–156, New York, NY, USA, 2010. ACM.

[74] Stuart Geman, Elie Bienenstock, and René Doursat. Neural networks and the bias/variance dilemma. *Neural Comput.*, 4(1):1–58, January 1992.

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

[76] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. *CoRR*, abs/1701.07232, 2017.

[77] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[78] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. ICSE'12.

[79] C. Le Goues, W. Weimer, and S. Forrest. Representations and operators for improving evolutionary software repair. GECCO'12.

[80] Ilya Grigorik. GitHub Archive. `https://www.githubarchive.org`, 2012.

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

[82] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 933–944, New York, NY, USA, 2018. ACM.

[83] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 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*, pages 631–642, 2016.

[84] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 631–642, New York, NY, USA, 2016. ACM.

[85] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deepam: Migrate apis with multi-modal sequence to sequence learning. *CoRR*, abs/1704.07734, 2017.

[86] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 3–14, Piscataway, NJ, USA, 2017. IEEE Press.

[87] Rahul Gupta, Aditya Kanade, and Shirish K. Shevade. Deep reinforcement learning for programming language correction. *CoRR*, abs/1801.10467, 2018.

[88] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. Deepfix: Fixing common c language errors by deep learning. In *AAAI*, 2017.

[89] R. G. Hamlet. Testing programs with the aid of a compiler. *TSE*, 3(4):279–290, July 1977.

[90] Z. Han, X. Li, Z. Xing, H. Liu, and Z. Feng. Learning to predict severity of software vulnerability using only vulnerability description. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 125–136, Sept 2017.

[91] Lile P. Hattori and Michele Lanza. On the nature of commits. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE'08, pages III–63–III–71, Piscataway, NJ, USA, 2008. IEEE Press.

[92] V. Hellendoorn and P. Devanbu. Are deep neural networks the best choice for modeling source code? FSE'17.

[93] Kim Herzig, Sascha Just, and Andreas Zeller. 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*, pages 392–401, 2013.

[94] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pages 121–130, 2013.

[95] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.

[96] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[97] Einar W. Høst and Bjarte M. Østvold. Debugging method names. In *ECOOP 2009 – Object-Oriented Programming*, Sophia Drossopoulou, editor, pages 294–317, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[98] Reyhaneh Jabbarvand and Sam Malek. μdroid: an energy-aware mutation testing framework for android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 208–219, 2017.

[99] John Jacobellis, Na Meng, and Miryung Kim. Lase: an example-based program transformation tool for locating and applying systematic edits. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1319–1322. IEEE Press, 2013.

[100] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, September 2011.

[101] S. Jiang, A. Armaly, and C. McMillan. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 135–146, Oct 2017.

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

[103] Magne Jorgensen and Martin Shepperd. A systematic review of software development cost estimation studies. *IEEE Trans. Softw. Eng.*, 33(1):33–53, January 2007.

[104] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.

[105] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 654–665, New York, NY, USA, 2014. ACM.

[106] Nal Kalchbrenner and Phil Blunsom. Recurrent continuous translation models. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1700–1709, Seattle, Washington, USA, October 2013. Association for Computational Linguistics.

[107] Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley-Interscience, 2005.

[108] Y. Ke, K. Stolee, C. Le Goues, and Y. Brun. Repairing programs with semantic code search. ASE'15.

[109] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 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*, pages 802–811, 2013.

[110] Sun-Woo Kim, John A. Clark, and John A. McDermid. Investigating the effectiveness of object-oriented testing strategies using the mutation method. *Softw. Test., Verif. Reliab.*, 11(3):207–225, 2001.

[111] P. Koehn. *Statistical Machine Translation*. 2010.

[112] Jacob Kogan. *Introduction to Clustering Large and High-Dimensional Data*. Cambridge University Press, New York, NY, USA, 2007.

[113] Carsten Kolassa, Dirk Riehle, and Michel A. Salim. A model of the commit size distribution of open source. In *SOFSEM 2013: Theory and Practice of Computer Science*, Peter van Emde Boas, Frans C. A. Groen, Giuseppe F. Italiano, Jerzy Nawrocki, and Harald Sack, editors, pages 52–66, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[114] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In

2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 476–481, Nov 2015.

[115] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 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*, pages 218–229, 2017.

[116] X. Le, D. Chu, D. Lo, C. Le Goues, and W. Visser. S3: Syntax- and semantic-guided repair synthesis via programming by examples. FSE'17.

[117] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 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*, pages 213–224, 2016.

[118] C. Le Goues, N. Holtschulte, E. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The manybugs and introclass benchmarks for automated repair of C programs. *TSE*, 41(12):1236–1256, 2015.

[119] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *TSE*, 38(1), 2012.

[120] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 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*, pages 3–13, 2012.

[121] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[122] Sun-Ro Lee, Min-Jae Heo, Chan-Gun Lee, Milhan Kim, and Gaeul Jeong. Applying deep learning based automatic bug triager to industrial projects. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 926–931, New York, NY, USA, 2017. ACM.

[123] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260, Sept 2017.

[124] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, June 1978.

[125] Mario Linares-Vásquez, Christopher Vendome, Michele Tufano, and Denys Poshyvanyk. How developers micro-optimize android apps. *Journal of Systems and Software*, 130:1 – 23, 2017.

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

[127] Qi Luo, Kevin Moran, Denys Poshyvanyk, and Massimiliano Di Penta. Assessing test case prioritization on real faults and mutants. In *Proceedings of the 34th International Conference on Software Maintenance and Evolution*, ICME '18, page to appear, Piscataway, NJ, USA, 2018. IEEE Press.

[128] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025, 2015.

[129] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: An automated class mutation system. *Software Testing, Verification & Reliability*, 15(2):97–133, June 2005.

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

[131] Matias Martinez and Martin Monperrus. Astor: A program repair library for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 441–444, New York, NY, USA, 2016. ACM.

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

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

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

[135] Haitao Mi, Zhiguo Wang, and Abe Ittycheriah. Vocabulary manipulation for neural machine translation. *CoRR*, abs/1605.03209, 2016.

[136] Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1):17:1–17:24, January 2018.

[137] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. Machine learning-based prototyping of graph-

ical user interfaces for mobile apps. *IEEE Transactions on Software Engineering*, abs/1802.02312:26, 2018.

[138] Kevin Moran, Michele Tufano, Carlos Bernal-Cárdenas, Mario Linares-Vásquez, Gabriele Bavota, Christopher Vendome, Massimiliano Di Penta, and Denys Poshyvanyk. Mdroid+: A mutation testing framework for android. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pages 33–36, New York, NY, USA, 2018. ACM.

[139] Samar Mouchawrab, Lionel C. Briand, Yvan Labiche, and Massimiliano Di Penta. Assessing, comparing, and combining state machine-based testing and structural testing: A series of experiments. *IEEE Trans. Software Eng.*, 37(2):161–187, 2011.

[140] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Statistical learning approach for mining api usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 457–468, New York, NY, USA, 2014. ACM.

[141] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 651–654, New York, NY, USA, 2013. ACM.

[142] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Migrating code with statistical machine translation. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 544–547, New York, NY, USA, 2014. ACM.

[143] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, and Hridesh Rajan. A study of repetitiveness of code changes in software evolution. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 180–190, Piscataway, NJ, USA, 2013. IEEE Press.

[144] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[145] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, 1996.

[146] A. Jefferson Offutt and Ronald H. Untch. Mutation testing for the new century. chapter Mutation 2000: Uniting the Orthogonal, pages 34–44. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[147] Oxford. How many words are there in the english language?, August 2018.

[148] Fabio Palomba, Dario Di Nucci, Michele Tufano, Gabriele Bavota, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Landfill: An open dataset of code smells with public evaluation. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 482–485, Piscataway, NJ, USA, 2015. IEEE Press.

[149] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA.*, pages 311–318, 2002.

[150] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[151] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[152] Terence Parr and Kathleen Fisher. Ll(*): The foundation of the antlr parser generator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 425–436, New York, NY, USA, 2011. ACM.

[153] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 87–102, New York, NY, USA, 2009. ACM.

[154] Derrin Pierret and Denys Poshyvanyk. 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*, pages 228–232, 2009.

[155] Lutz Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, pages 55–69, London, UK, UK, 1998. Springer-Verlag.

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

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

[158] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428, New York, NY, USA, 2014. ACM.

[159] Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann. *Recommendation Systems in Software Engineering*. Springer Publishing Company, Incorporated, 2014.

[160] S. Romansky, N. C. Borle, S. Chowdhury, A. Hindle, and R. Greiner. Deep green: Modelling time-series of software energy consumption. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 273–283, Sept 2017.

[161] X. Rong. word2vec parameter learning explained. *CoRR*, abs/1411.2738, 2014.

[162] Alexander M. Rush, Sumit Chopra, and Jason Weston. A neural attention model for abstractive sentence summarization. *CoRR*, abs/1509.00685, 2015.

[163] S. Scalabrino, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto. Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016.

[164] David Schuler and Andreas Zeller. Javalanche: efficient mutation testing for java. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, pages 297–298, 2009.

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

[166] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 201–211. IEEE, 2015.

[167] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. *SIGPLAN Not.*, 50(6):43–54, June 2015.

[168] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 532–543, New York, NY, USA, 2015. ACM.

[169] Victor Sobreira, Thomas Durieux, Fernanda Madeiral Delfim, Martin Monperrus, and Marcelo de Almeida Maia. 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*, pages 130–140, 2018.

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

[171] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[172] Yuchi Tian and Baishakhi Ray. Automatically diagnosing and repairing error handling bugs in c. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 752–762, New York, NY, USA, 2017. ACM.

[173] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 329–331. IEEE, 2008.

[174] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, 43(11):1063–1088, Nov 2017.

[175] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 4–15, New York, NY, USA, 2016. ACM.

[176] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4):e1838, 2017. e1838 smr.1838.

[177] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 403–414, Piscataway, NJ, USA, 2015. IEEE Press.

[178] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, 2019.

[179] Michele Tufano, Hitesh Sajnani, and Kim Herzig. Towards predicting the impact of software changes on building activities. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, 2019.

[180] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Deep learning similarities from different representations of source code. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 542–553, New York, NY, USA, 2018. ACM.

[181] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 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*, ASE 2018, pages 832–837, New York, NY, USA, 2018. ACM.

[182] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *CoRR*, abs/1812.08693, 2018.

[183] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Learning how to mutate source code from bug-fixes. *CoRR*, abs/1812.10772, 2018.

[184] Reel Two. Jumble. `http://jumble.sourceforge.net`.

[185] Danny van Bruggen. JavaParser. `https://javaparser.org/about.html`, 2014.

[186] Mario Linares Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Enabling mutation testing for android apps.

In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 233–244, 2017.

[187] Oriol Vinyals and Quoc Le. A neural conversational model. *arXiv preprint arXiv:1506.05869*, 2015.

[188] Kaiyuan Wang. *MuAlloy: an automated mutation system for alloy*. PhD thesis, 2015.

[189] Song Wang, Taiyue Liu, and Lin Tan. 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*, pages 297–308, 2016.

[190] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 297–308, New York, NY, USA, 2016. ACM.

[191] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 356–366, Piscataway, NJ, USA, 2013. IEEE Press.

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

[193] Aaron Weiss, Arjun Guha, and Yuriy Brun. Tortoise: Interactive system configuration repair. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 625–636, Piscataway, NJ, USA, 2017. IEEE Press.

[194] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 1–, Washington, DC, USA, 2007. IEEE Computer Society.

[195] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 479–490, Feb 2019.

[196] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 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*, pages 87–98, 2016.

[197] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 334–345, Piscataway, NJ, USA, 2015. IEEE Press.

[198] E. Wong, Jinqiu Yang, and Lin Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 562–567, Nov 2013.

[199] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's

neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[200] Zhenchang Xing and Eleni Stroulia. Api-evolution support with diff-catchup. *IEEE Transactions on Software Engineering*, 33(12):818–836, 2007.

[201] J. Xuan, M. Martínez, F. DeMarco, M. Clément, S. Lamelas, T. Durieux, Daniel Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in Java programs. *TSE*, 43(1):34–55, 2016.

[202] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 831–841, New York, NY, USA, 2017. ACM.

[203] Ying Zhang, Stephan Vogel, and Alex Waibel. Interpreting bleu/nist scores: How much improvement do we need to have a better system. In *In Proceedings of Proceedings of Language Resources and Evaluation (LREC-2004*, pages 2051–2054, 2004.

[204] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining api mapping for language migration. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 195–204, New York, NY, USA, 2010. ACM.

[205] Thomas Zimmermann, Nachiappan Nagappan, Harald C. Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, pages 91–100, 2009.

[206] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.